Automated Attacker Correlation for Malicious Code

Thomas Dullien, Ero Carrera, Soeren-Meyer Eppler, Sebastian Porst

Ruhr-University Bochum
zynamics GmbH
Grosse Beck Str 3.
44787 Bochum
Germany

March 22, 2010

Abstract

Correlating attacks can be specifically problematic in the digital domain. It is a common scenario that the only real "trace" of an attack that can be obtained is executable code. As such, executable code of malicious software forms one of the primary pieces of evidence that need to be examined in order to establish correlation between seemingly independent events/attacks.

Due to the high technical sophistication required for building advanced and stealthy persistent backdoors ("rootkits"), it is quite common for code fragments to be re-used. A big obstacle to performing proper correlation between different executables is the high degree of variability which the compiler introduces when generating the final byte sequences.

This paper presents the results of research on executable code comparison for attacker correlation. Instead of pursuing a byte-based approach, a structural approach is chosen. The result is a system that can identify code similarities in executables with accuracy that often exceeds that of a human analyst and at much higher speed.

1 Introduction

One of the core problems facing actors in the cyber domain is the difficulty of attribution. While this problem appears as hard as ever, a related problem is "correlation" – even if it is unclear *what specific actor* is behind an attack, it is desirable to *group attacks by suspected actor*. This allows much improved intelligence on and understanding of coordinated multi-pronged attacks.

Often, the principal artifact of a successful attack that the defenders can perform analysis of is malicious code that was obtained from compromised systems. Direct comparison of such artifacts is difficult due to the heavy changes that are introduced by the compilation and optimization process.

The problem of comparing executables derived from a common source code origin but compiled using different compiler settings was first studied in the context of *security patch analysis*. In this scenario, two versions of the same executable that were compiled on similar but nonidentical build environments are to be compared, and the security implications of a software update is then examined. The executables are usually relatively similar, but variability on the byte level exists nonetheless. Motivated by this, structural approaches to compare executables are discussed in [6, 7, 9, 3]. In [1, 2], applications of such methods to malware classification were given. Further work along similar lines was recently performed by [8].

2 Contribution

The primary research contributions of this paper are the following:

1. A comparison algorithm that allows comparison of executable code independent of compiler optimizations. This algorithm is an extension of the algorithms discussed in [7, 3] and has been successfully used to compare code compiled for different operating systems (using different compilers) [5] and different architecture [4].

2. A full system that, based on the above algorithm, allows the fast comparison and correlation between different pieces of malicious software.

3. An algorithm that allows for specific efficient queries for particularly interesting functions into a large database of functions.

Two practical case studies are also studied – both originating in a real-world computer network attack. Further work will be presented where correlation is done amongst large quantities of malicious software.

3 Structural Executable comparison

Compilers enjoy wide-ranging freedom when assigning instruction sequences and registers to perform particular tasks. This leads to a wide variability of the byte sequences that are generated from a particular piece of source code. A compounding problem arises from the fact that many compilers use non-deterministic and randomized algorithms to perform common tasks such as register allocation: Even identical build environments and compilers can generate different byte sequences due to this.

On the structural level, the freedom is much more restricted: The control flow logic of the underlying source code dictates, to a certain extent, the control flow logic of the emitted assembly, and thus the "shape" or topology of the control flow graph. While the compiler has a certain level of freedom in creating control flow graphs, the need to generate efficient code leads to a 'convergence' to a few possible shapes.

Techniques for using the structure of the control flows for executable comparison were pioneered independently by [9] and Flake's [6]. The latter of the two already includes measures that also take the callgraph structure of the executable into account. Callgraph similarity measures were used by [1] to automatically recognize similarities between variants of the same malicious software.

4 The goal

The goal for correlation of malicious software is two-fold:

1. Allow specific querying for a particular structural feature (control flow graph) in a large body of malicious code

2. Automatically recognize similarities between superficially different pieces of malicious code

In the presented work, the first goal is achieved by identifying a way to perform quick database lookup for a particular graph structure. The second goal is achieved by a more involved algorithm that computes similarities between two executables based on their structural properties.

5 Fast database lookup for flowgraphs

In order to allow fast querying into large sets of data, it is desirable to implement something like a *hash function* for control flow graphs. Given an easily-calculated encoding of control-flow-graphs into a sequence of bits, fast lookup is possible using either hashtables or binary search trees (as the encoding automatically imposes a total ordering on the space of control flow graphs).

The question then becomes: What is a good and fast way to encode a control flow graph into a (possibly small) sequence of bits, if possible with a low probability of random collisions (e.g. same hash values for structurally different flowgraphs) ?

We begin by converting the set of edges in a control flow graph into a set of n-tuples of integers. In the next step, a suitable encoding for this set of n-tuples is constructed which attempts to minimize the odds of multiple sets of tuples mapping to the same value.

The construction of this set of tuples works as follows: Let $\mathfrak{G}$ be the set of all control flow graphs and E_g the set of edges of a particular $G \in \mathfrak{G}$. We then define:

$$tup : \mathfrak{G} \rightarrow \mathfrak{P}(\mathbb{Z}^5)$$

$$tup(g) \mapsto \left\{ \begin{pmatrix} \mathsf{topologicalorder}(src(e)), \\ \mathsf{indegree}(src(e)), \\ \mathsf{outdegree}(src(e)), \\ \mathsf{indegree}(dest(e)), \\ \mathsf{outdegree}(dest(e)) \end{pmatrix} | e \in E_g \right\}$$

This provides a set of integers from a given graph G in a quite straightforward manner. The next question is now the choice of a suitable encoding that minimizes unwanted collisions. In order to achieve this, we notice that these tuples form a subset of vectors from $\mathbb{Q}^5$ and the fact that we can *embed* the vector space $\mathbb{Q}^5$ (which is a 5-dimensional vector space over $\mathbb{Q}$) into the vector space $\mathbb{Q}[\sqrt{2}, \sqrt{3}, \sqrt{5}, \sqrt{7}]$. This vector space is also a 5-dimensional vector space over $\mathbb{Q}$ with the added property that each element of this vector space is *also* a real number. We thus have an easy way of converting a 5-tupel into a real number:

$$emb(z) \mapsto z_0 + z_1\sqrt{2} + z_2\sqrt{3} + z_3\sqrt{5} + z_4\sqrt{7}$$

The only thing that we still need to construct our hash function is now a method to combine the values obtained from the individual tuples in a way that is agnostic of the ordering of the edges while at the same time avoiding unwanted side effects. The final construction is the following:

$$Hash(g) := \sum_{t \in tup(g)} \frac{1}{\sqrt{emb(t)}}$$

In reality, all calculations happen over floating point numbers and not over $\mathbb{R}$ – e.g. the mathematical intuition above which works decently over $\mathbb{R}$ is reduced to handwaving in real implementations. Nonetheless, empirical evidence from several hundred thousand control flow graphs has shown that the odds of structurally different control flow graphs randombly hashing to the same value is exceedingly low and does not matter in practice.

In the following, we call this hash function the *MD-Index* for a given graph.

6 Comparing full executables

A fast lookup of individual control flow graphs is just one part of a proper correlation system. Compiler optimizations often have a pretty significant impact on the structure of control flow graphs, and more thorough comparison algorithms that can deal with small modifications of control flow

graphs are desirable. These algorithms should exploit properties of the callgraph structure along with control flow graph structures, and also take miscellaneous other properties (use of common API functions etc.) into account.

6.1 The Goal

In order to determine the extent of code sharing between two executables, a mapping between the functions, basic blocks, and instructions in the two executables is constructed. This means that the algorithm constructs a mapping that for each function/basic block/instruction in the first executable contains a reference to the corresponding function/basic block/instruction in the second executable (if such a corresponding element exists).

In order to construct these mappings, the algorithm approximates a solution for the maximum subgraph isomorphism problem on both the level of the executable callgraphs and the executable flowgraphs.

Specifically, this means that an approximation of the maximum subgraph isomorphism is constructed between:

1. The two callgraphs of the two given executables
2. Each pair of flowgraphs that arise from the matching of two nodes in the two respective callgraphs

6.1.1 The algorithm

The algorithm that is used to construct the approximation to the maximum subgraph isomorphism problem is identical on both the callgraphs and flowgraphs - some details change, but the algorithm stays identical. Let $G = (N, E), G' = (N', E')$ be the graphs in question.

Definition 1 (Characteristic) *A pair of mappings $\sigma : N \to D, \sigma' : N' \to D$ from the set of nodes in a graph to some domain D is called a* node characteristic. *A pair of mappings $\sigma : E \to D, \sigma' : E' \to D$ is called an* edge characteristic. *To simplify notation, σ will be written for both σ and σ'.*

We maintain a separate list of characteristics for both the callgraph and the control-flow-graph - e.g. the only difference between the algorithm on the callgraphs vs. on the flowgraph is in the list of characteristics.

The algorithm itself is relatively simple:

Given an arbitrary node, each characteristic provides a set of *candidates* that might be a good match for this node. The characteristics are applied successively to refine the set of candidates down to a unique match; if a unique match is found, the nodes are associated. If no unique match is found by refining from the first characteristic (usually because no match is found at all) the first characteristic is stripped off the list, and the process is iterated.

Data: N, N' (sets of either edges or nodes), List of (node or edge) characteristics $\sigma_1, \ldots, \sigma_m$
Result: Dictionary D that contains mappings N to N'
for $i \leftarrow 1$ **to** m **do**
  foreach $n \in N$ **do**
    $temp \leftarrow \{n' \in N' | \sigma_i(n') = \sigma_i(n)\}$;
    for $j \leftarrow i$ **to** m **do**
      if $|temp| = 1$ **then**
        $D[n] = temp$;
        break;
      end
      $temp \leftarrow \{n' \in temp | \sigma_j(n') = \sigma_j(n)\}$;
    end
  end
end

Algorithm 1: Calculating a mapping given node or edge characteristics

The values for $\sigma_i(n')$ can be precomputed. If D is "cheaply" ordered (e.g. allows constant-time comparison of elements) and the values for $\sigma_i(n')$ were precomputed, lookup of elements for the line $temp \leftarrow \{n' \in N' | \sigma_i(n') = \sigma_i(n)\}$ can be done using a regular binary tree - the computational cost is hence logarithmic in $|N'|$. A quick analysis of the algorithm suggests that the runtime complexity if all values are precomputed should be upper-bounded by

$$O(m^2 |N| \log(|N'|)^2)$$

There is no backtracking involved in this algorithm, and no complicated decisions are made. Speed is given precedence over accuracy, and the accuracy is obtained through the choice of characteristics. The exact characteristics will be discussed in section 6.2.

6.2 List of characteristics

From the previous section it becomes clear that the proper choice of characteristics is crucial for proper functioning of the discussed algorithm. In the following, we list some criteria that are used for the graph matching step, both on the control flow graphs and on the callgraphs.

Byte Hash A traditional hash over the bytes of the function or the basic block is calculated.

MD-Index of a particular function For a given node in the callgraph, the MD-Index of the control flow graph of the underlying function is constructed. This can only be applied to callgraph nodes.

MD-Index of source and destination of callgraph edges For a given edge in the callgraph, the MD-Index of the control flow graphs of the underlying functions for both the source and the destination of the edge is calculated. The tuple consisting of two values forms the characteristic for the edge.

MD-Index of graph neighborhood of a node/edge For a given node (or edge), a subgraph of the larger graph consisting of a neighborhood of the original graph of a given size is extracted (for example, all nodes that are less than 2 "hops" away). The MD-Index of this "local" graph is calculated. This characteristic has the interesting property that nodes are matched not by their intrinsic properties (e.g. the code contained therein), but by the *local structure* of the graph of which they are part. This means that even in situations where both the callgraph and the individual control flow graphs changed significantly, proper matching is still possible.

Small Prime Product The *small prime product* [3] is a simple way of calculating a hash of a sequence of mnemonics that ignores compiler-induced reordering of instructions. Each mnemonic is mapped to a small prime number, and the product of all elements in the sequence is calculated (module 2^{64}).

More characteristics (hashes of string references etc.) can be added easily. It is notable though that characteristics that generate few "collisions" (e.g., that on average map few elements to each image value) produce better matchings in practice than those that produce many "collisions".

7 Full system implementation

The described algorithms have been implemented in an automated system to process large volumes of malicious software. The system is designed around a central database from which multiple different workers fetch data to perform computations on. This allows easy distribution of the computationally intensive parts.

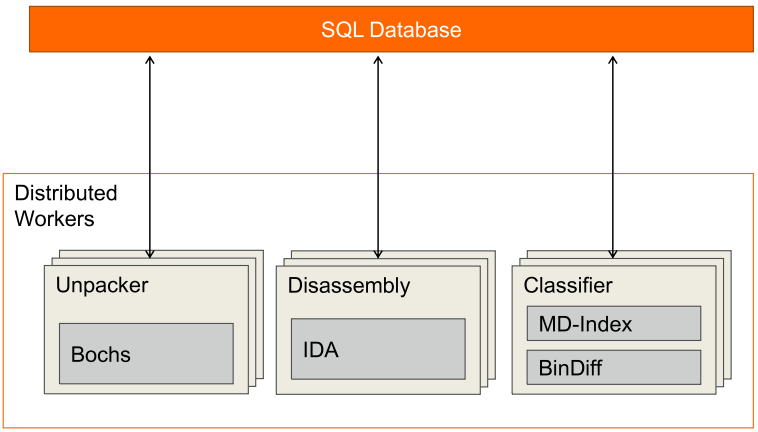

Figure 1: Architecture: A central database server that many workers poll from/write to

The system consists of a non-distributed scheduler component, and distributed components called *Unpacker, Disassembly* and *Comparison*. These components are described below:

Unpacker: An unpacking components attempts to remove executable encryption by emulating the executable while monitoring the statistical properties of the emulated system RAM. Once the entropy of the memory pages drops, the component assumes that encryption/compression was removed, and writes memory dumps back into the central database.

This step in the processing of malware can be skipped if memory dumps were obtained in a different manner.

Disassembly: A disassembly component disassembles a memory dump and extracts the control-flow-graph structures from the disassembly. Functions are identified, the callgraph is extracted, and the results are written back to the central database.

Scheduler: Performing a full comparison of all files amongst each other would by its very nature be quadratic, and hence prohibitively expensive.

In order to reduce the overall cost, the scheduler performs a rough comparison based on MD-Indices of functions in the disassembly. This cheap comparison is used to *schedule* more expensive comparisons: For a new executable, accurate comparisons against those existing

executables are scheduled that have a non-negligible similarity score in the approximate comparison.

Comparison: A comparison-worker queries the database for pairs of executables to be compared using the "more accurate" algorithms described below. It then fetches the relevant data from the database, performs the comparisons, and writes the results of the comparison back to the database.

8 Case studies

The system was used for attacker correlation in several real-world scenarios. In the following, we will describe two real-world cases, followed by some internal evaluation.

8.1 Attack on financial services provider

Around the same time the much-publicized "Aurora" attacks appeared in the media, we were contacted by one of our clients in the financial services sector. Several machines on their network had been compromised, and malicious software had been found on these systems. Due to the timing and widespread media attention given to "Aurora", the customer suspected that the attacks that they had fallen victim to were part of the same attacker.

In order to confirm or disprove this suspicion, the system was fed with a number of executables that had been obtained from the compromised machines on the clients network, along with executables that were part of the "Aurora" attacks. Some manual intervention was required to remove the obfuscation layers from the executables – since they were loaded in a nonstandard manner, the generic emulation could not handle them straight away.

The result of the comparison/classification process showed that the executables obtained from the compromised machines were all very similar amongst each other - sharing between 98% and 99% of all the code.

On the other hand, no discernible similarity between these files and various files involved with the "Aurora" attacks were found.

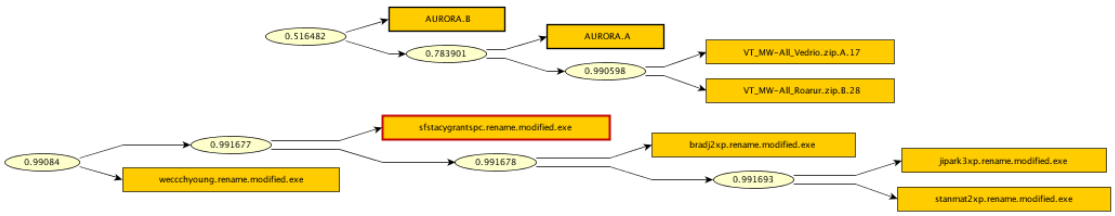

Figure 2: Case study: Classification results showing no similarity

8.2 Sophisticated Rootkit

We were contacted by the victim of a large-scale, multi-year attack on their network infrastructure to perform an analysis of the rootkit code that had been obtained on several of their hosts. During the investigation, many versions of the rootkit code were uncovered. Most of them were relatively recent. On some legacy systems, a suspicious device driver that dated back several years was found. This driver did not exhibit a superficial similarity to the "current" version of the rootkit.

Nonetheless, a structural comparison as described above determined that about 48 % of the code from the new version of the driver was similar to (or derived from) the old version of the driver. This helped tremendously in establishing important points on the timeline of the attack.

8.3 Large-scale clustering

To perform further tests, 5000 randomly chosen malicious executables were automatically grouped into families using the similarity metric above – executables that shared more than 60 % of their overall code were considered one cluster.

From these 5000 executables, a few of the larger clusters were extracted. For each cluster, names were determined by querying the VirustTotal database – if a few executables in a cluster were detected with consistent names by *any* anti-virus product, the clusters were named accordingly.

Name of the cluster	# of executables
Win32.KillAV.Variants	183
Win32.Bacuy.Variants	599
Win32.SkinTrim	173
Win32.SwizzorA	15
Win32.WinTrim	114
FakeAlert	54
Win32.Allaple	39
Win32.Prosti	27
Win32.Sality	21
Win32.Zhelatin?	22
Win32.Swizzor?	14
Win32.Chifrax	12

Table 1: The number of executables for the different clusters

9 Summary

Automated correlation between attackers is possible based on *sharing of executable code* which, in turn, is possible through the structural comparison algorithms described in this paper. Shifting the focus from byte-sequences to the graph structure of the executables (and keeping both the control-flow-graph structure as well as the callgraph structure in mind) allows the correlation of software in the presence of heavy compiler-induced changes to the bytecode.

References

[1] Ero Carrera and Gergely Erdelyi. Digital genome mapping - advanced binary malware analysis. In *Proceedings of the Virus Bulletin Conference 2004*, pages 187–197, 2004.

[2] Ero Carrera and Halvar Flake. Automated structural classification of malware. In *Proceedings of the RSA Conference 2008, 7-11 April, 2008*, 2008.

[3] Thomas Dullien and Rolf Rolles. Graph-based comparison of executable objects (english version). In *SSTIC '05, Symposium sur la Sécurité des Technologies de l'Information et des Communications, June 1-3, 2005, Rennes, France*, 06 2005.

[4] Halvar Flake. Diffing x86 vs arm code.

[5] Halvar Flake. Improving binary comparison.

[6] Halvar Flake. More fun with graphs. In *Blackhat Federal 2003*, 2003.

[7] Halvar Flake. Structural comparison of executable objects. In Ulrich Flegel and Michael Meier, editors, *DIMVA*, volume 46 of *LNI*, pages 161–173. GI, 2004.

[8] Xin Hu, Tzi-cker Chiueh, and Kang G. Shin. Large-scale malware indexing using function-call graphs. In *CCS '09: Proceedings of the 16th ACM conference on Computer and communications security*, pages 611–620, New York, NY, USA, 2009. ACM.

[9] Todd Sabin. Comparing binaries with graph isomorphism. 2003.