Benign Software Corpus

Brian Beisel Moshe Katz Yehuda Katz

With new variants of malware coming out every day, Anti-Malware applications have an increasingly difficult job detecting malware and protecting computers from it. It is estimated that more than 1 million new malware variants are produced every single day. However, the number of benign software programs a legitimate user may wish to run is far lower. We propose the creation of a benign software corpus, and the collection of statistical data about the attributes of the files in the corpus. Comparison of unknown or untrusted hosts and/or the executable files they contain with the data in this corpus will allow Anti-Malware applications to determine that a file is likely benign, a much easier task than determining whether the file is likely malicious.

Introduction

There is no question that executable malware is a significant threat to all computer users today. The sheer number of malware programs available online is staggering; some masquerading as regular "safe" software, some as illicit "tools" like License Key generators or "Cracks" for paid software, some as hacking tools, and some not even trying to hide its evil. The task of detecting all such malware is quite difficult and there will likely always be new malware samples that skip through the detection mechanism. This detection is especially difficult given that security experts estimate hundreds of thousands (Higgins, 2012) or even millions (Dumitras, 2013) of new malware samples are released every single day! That is certainly significantly more, likely several orders of magnitude more, than the number of benign executable files released every day. Although many of these new malware samples are detected by existing Anti-Malware definitions and heuristics, it is fairly obvious that the malware authors are ahead of the game. As long as malware authors maintain this lead, it will be extremely difficult to ensure that all malware samples are caught and neutralized.

In recognition of this danger, many Anti-Malware vendors are turning to white-list or reputation based systems to determine whether a file is "good" instead of whether a file is "bad." Services like Microsoft's SmartScreen, Google's Safe Browsing, and Symantec's Polonium use reputation-based systems in addition to traditional malware scanning to help determine whether a file is safe or not. While these techniques add a significant layer of additional protection, by warning against (or outright blocking) execution of files with low reputation, reputation-based protection still has a high false-positive rate for legitimate files that just don't have a good reputation yet.

Instead of trying to detect all malware or to calculate the reputation of arbitrary executable files, we postulate that it may be possible to build metrics that identify safe, or "benign," software, based on attributes of the benign executable files themselves. However, before this theory can be studied, we need to collect and analyze a corpus of executables to determine what properties, or "metadata," may be indicators of the benignity of the files in the corpus. Our goal in this study is to begin the collection of such a corpus, and to determine whether this corpus is useful and whether keeping it up to date is feasible.

Data Collection

Data Sources

The most important building block of a software metadata corpus is, of course, a large collection of metadata of ***known-benign*** software. It was therefore necessary to evaluate several options of data sources, in order to determine which would best fit the needs of this project.

The first, and also the largest, data source we considered was the National Software Reference Library (NSRL), maintained by the National Institute of Standards and Technology. The NSRL is designed "to collect software from various sources and incorporate file profiles computed from this software into a Reference Data Set (RDS) of information [which] can be used by law enforcement, government, and industry organizations to review files on a computer by matching file profiles in the RDS" (NIST, n.d.). The NSRL contains data about more than 114 million files, including SHA-1, MD5, and CRC32 hash values, file names, application categories, manufacturer names, product names, and operating system versions. However, we decided against using this data source for several reasons. First, other than application categories and manufacturers, this dataset does not actually provide significant metadata about programs in it.[1] Second, this dataset includes a lot of noise, in the form of non-executable files, executables for non-Windows operating systems, and executables that run only on older versions of Windows no longer supported or in common use. Third, inclusion in the NSRL dataset is solely based on the decision of NIST, which can choose not to include a piece of software. Additionally, most of the software included in the NSRL dataset comes from manufacturer submissions of "shrink-wrapped Common-Off-The-Shelf (COTS)" software, which does not include updates downloaded from the Internet – which we anticipate seeing a lot of on normal computers. Finally, the NSRL includes data about some programs which they say "**may be considered malicious**," (emphasis theirs) necessitating processing of the dataset to remove those entries.

We also considered using ShadowServer's Bin Check service, but rejected it for similar reason to the NSRL. This is unsurprising, given that a large portion of ShadowServer's data comes from the NSRL. ShadowServer does include additional metadata, such as digital signatures, and directory names. However, ShadowServer does not handle the possibility of a file legitimately occurring in multiple directories and provides only a single directory entry for each file.

"SoftwareScanner" Client

In the end, we determined that we would get the best results by scanning for all executable files on computers that we had a reasonable expectation were trustworthy and contained only benign software. This allowed us to choose exactly what metadata points we wanted to capture.

Our program is called SoftwareScanner, and it is targeted to run on trusted computers running Windows Vista or newer. SoftwareScanner scans the local hard drive (or a list of user-selected locations) for all files in the Portable Executable ("PE") format, calculates their hashes, collects their metadata, and sends

[1] While it is important to collect the hash data, hashes are, by definition, not analyzable to determine whether software is benign, and therefore not significant metadata for us. Also, NIST-assigned "categories" are equally useless to us, being arbitrary designations given by NIST that do not translate to analysis of non-labeled software.

the hashes and metadata to our server. For analysis purposes, it also submits the Windows version and bitness and a unique ID that identifies the scan and marks all files found in that scan.[2]

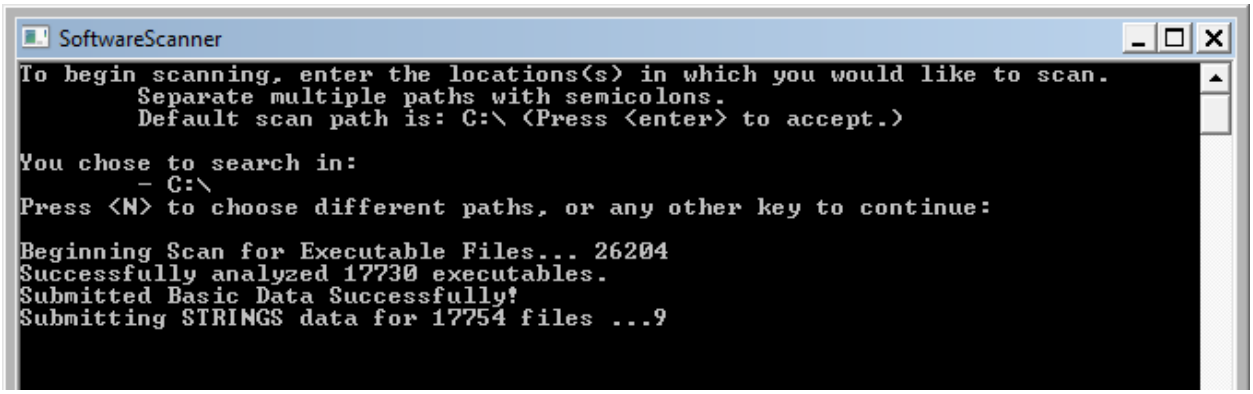


Figure 1 SoftwareScanner in action. Note that the scan found 26,204 files, but only analyzed 17,730 of them. That means there were 8,474 duplicate executables on this computer.

Data Elements

Since we were collecting our own data, we needed to determine what attributes we wanted to collect from each of the files we scanned. There are a lot of possible data elements that could be considered "metadata" of an executable file. In the end, we decided to collect the ones listed in Figure 2 below.

File Name []	Size
Full Path []	"From the Internet?"
Date Created / Modified	"Strings" []

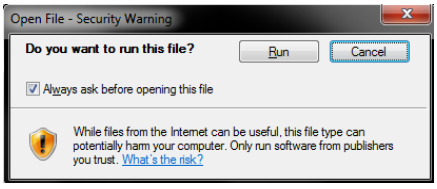
Figure 2 List of data elements collected

Notes About Specific Elements

File Name and Path. While still on the scanned computer, files found were grouped by SHA-1 hash to identify multiple instances of the same file with different names and/or folder paths. They were then submitted to the server as a record of a single file with multiple locations.

File Extension. Although traditional Windows executables use the file extension ".exe", we scanned for all files using any of the eight file extensions typically used to represent PE-format files. We did this to ensure that we collected data on executable DLLs (which are executed using Windows' 'rundll32' functions), screen savers, and device drivers, all of which are valid Windows executables.

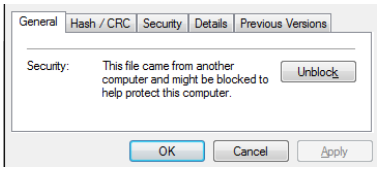
"From the Internet" refers to Microsoft's way of identifying that a file has been downloaded from the Internet and the user should be prompted every time it runs.[3] It is uses a simple text flag, hidden in an NTFS "Alternate Data Stream" that is stored on the disk along with the



[2] While SoftwareScan currently collects the real machine name of each scanned computer, that data is collected only for debugging the scanner application and is not used for any scans. Any future public version of this tool would not collect such identifiable data and would instead generate a random Machine ID.

[3] This is the same data that triggers "protected mode" when opening documents from the Internet in newer versions of Microsoft Office.

downloaded file that identifies that the file came from the internet, local network or possibly other locations. The open file confirmation dialog and the file properties box give the user a way to remove the flag.



"Strings" refer to actual text content inside the executable. They were collected using the SysInternals Strings utility. To avoid licensing issues should we choose to distribute SoftwareScan, we did not bundle the Strings executable, instead prompting the user to download it when necessary.

Scan Results

Using our scanning tool, we collected metadata for 123,794 unique executable files (as counted by SHA-1), totaling close to 24 GB of data.

The vast bulk of our data by disk space is the strings. Collecting all of the Strings all at once for a computer with 20,000 executable files takes over 10GB of RAM to process. Therefore, we collected Strings as a second step, only after the primary data submission was complete and the server could respond with a list of which files had not yet had Strings submitted. We also submitted the strings to the server one executable at a time to prevent excessive memory usage on both sides. This process was extremely slow, so we did not have time to complete the string collection for our entire dataset. We currently have string data for 62% of collected executables.

All scan results were submitted via HTTPS to a server owned by the University of Maryland chapter of the Association for Computing Machinery, collocated at the Department of Computer Science. They were processed by a PHP script and inserted into a MongoDB database.

Data Processing

Technical Requirements

Access. The collected data must be easily and quickly accessible to any person or program that needs it. Some of the processing done may only require a subset of our dataset. We stored our data in a MongoDB database which is not accessible directly. Instead, the client makes a query and downloads the resulting data file to use in the analysis application. This ensures that the analyzer is responsive (because it does not have to fight for database server resources) and that the data integrity of the database is not compromised by outside access (due to MongoDB's lack of proper authentication and authorization mechanisms).

Processing. Applications must be able to perform analysis of the collected data. The difficulty with our massive dataset is that it cannot fit into memory at once on any hardware we have access to. The solution is for the application to request a batch, or smaller group, of data at one time. The program can then process the data, save the results into a partial or compressed state, then release the batch and request for another. While the garbage collector often takes a long time to successfully release the unreferenced memory, if the partial state size is acceptably bounded, then the application will be guaranteed to be able to finish running. We implemented a data provider that fulfills this exact function.

Processing Application User Interface

Approach. Large datasets of multivariable descriptors can be difficult to analyze effectively since there are endless ways to view and compare the data. It is therefore important to construct an application

that is able to process given data with any desired configuration and immediately display the results. This way, a researcher can observe and compare different aspects of multiple datasets in a short amount of time and can easily process new data.

Design. One or two data files attained from our database server can be processed at once. The dataset we constructed can be grouped by machine, so our application is able to analyze and compare any two subsets of machines at one time. Once the data is selected, display and processing options are available to the user which dictate what analysis is done and how the results are shown. These options are essential to successful program execution due to existing display and processing constraints. The granularity of the partial and compressed state can be changed depending on the size of the dataset to be processed (Figure 3: Data Options -> Bin Size in KB[4]). The larger the dataset, the more the partial state must be compressed in order to avoid running out of application memory. The datasets used in this paper are orders of magnitude smaller than what is needed in practice but even they are large enough to grow an uncompressed state beyond application memory limits.

The application has two areas to display analysis results. A tabbed graphical area is used to display any charts and figures that are dynamically created by the analysis. There is a console output area where text results can be written to and are categorized for easier access. Our application features of data selection and display allow our large datasets to be observed with a relatively small amount of time and effort.

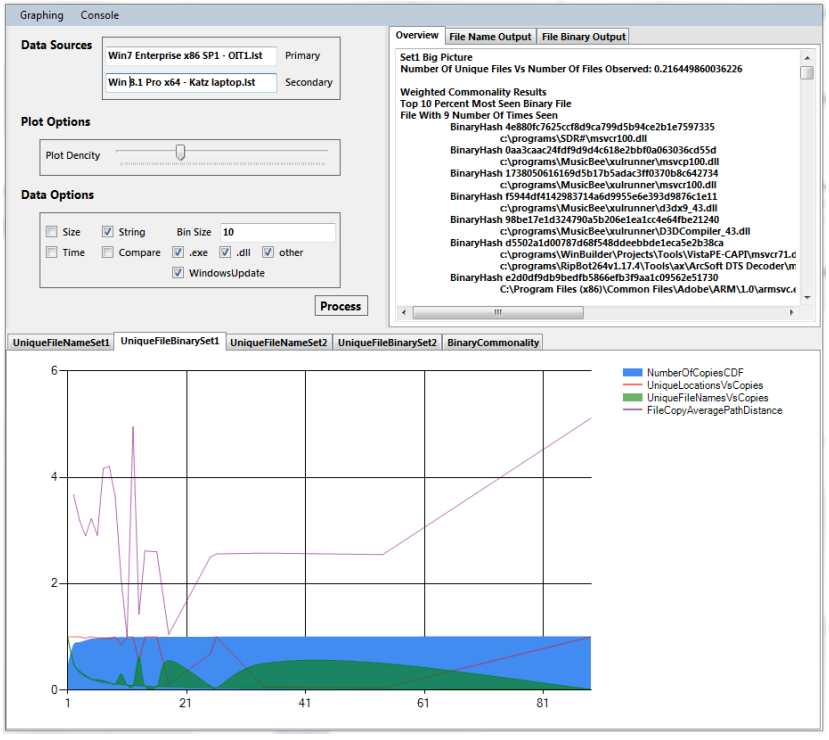


Figure 3 Data Processing Application User Interface

[4] When graphing something with respect to file size, each interval of bin size must be grouped together and its data compressed. This is not an issue with graphing with respect to variables with a higher per point density because then the intermediate state is adequately compressed automatically.

Data Analysis

Initial Look

The goal of our data analysis is to find characteristics and patterns of benign software. Once patterns are found, it is important to find the reasons behind them in order to create the logic around determining if a given file is benign, which of course is the entire point. To start, we got a view of what the files by themselves looked like. Looking at size we could see that the vast majority of binary files were quite small with 60 percent of files under .167MB and 90 percent under 1MB. However, this distribution did not tell us much about the characteristics of benign data other than the fact that a 60MB binary file is quite out of the ordinary. Looking at creation and access times did not show a useful distribution. Our data collection was not implemented to keep track of all the different times each file was created so that was not a possible analysis however interesting it may have proven to be. In order to find patterns, the data must be grouped in some effective way. Since generic data analysis groupings were proving fruitless, something else must be used.

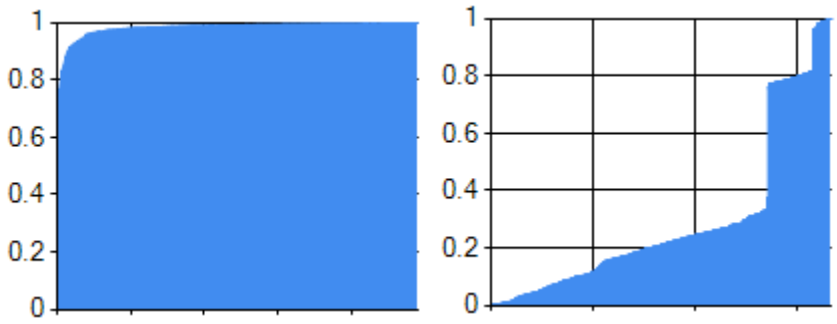


Figure 4 Size then Time CDF of Full Collected Dataset

Improved Groupings

Files were then grouped by identical file names and identical binaries. This gave way to several much more interesting ways to observe the data.

Each file name in our dataset was mapped to the total number of times it had been used to name a file. Files were then grouped with other files whose name had also been used that same number of times. The total number of files seen in each group plus the total number of files in the groups prior divided by the total number of files in the data was graphed. This gives us the cumulative distribution of file names based on the uniqueness of their names (Figure 5: Blue area). For all the binary executables of type '.exe' only 16 percent have names that are only used by one file. Files whose names are used one or two times make up 26 percent of the total files seen. By the time 50 percent of our executables have been seen we are up to including all names repeated up to 7 times. From this we can see that repeated file names are quite common. Considering that our data is only from a handful of computers, it is quite probable that a much larger dataset will have a much smaller fraction of names that have not been seen. Since average possibility space for file names (accounting average file name length) is significantly larger than the rate at which the number of unique file names decreased here it is safe to assume that benign file names have some pattern for which they are expressed. This makes a lot of sense since many of the names are made by humans who are bound by nature to some form of organization. Some of the most common names found are setup.exe, win32k.sys, and iexplore.exe.

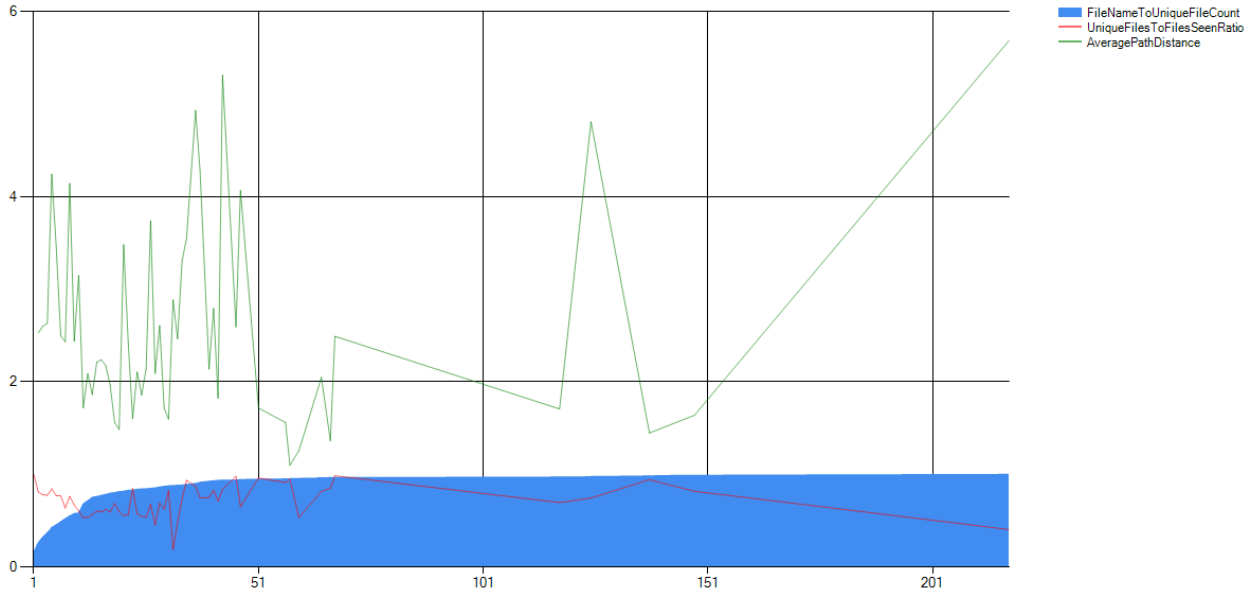


Figure 5 File Name Grouping

We can look at the ratio of files with the same name compared to how many unique binaries do those files represent (Figure 5: Red Line). When the ratio is close to 1 it indicates that the files named the same all represent different binaries while when the ratio is close to 1/n if means that they all represent a single binary. For each group of files with repeated names we can look at the graph or file structure distance away from each other. We can calculate all possible differences for each file name and average those distances in to a single number for each repeat group (Figure 5: Green Line). This number paired with the unique representation ratio can indicate more about the layout of the average benign file system. It seems to correlate that when a file name represents the same binary, the average distance between those files is shorter. Conversely, when the name represents all different binaries, their average distance is greater. It would only make sense that identical binaries would be found closer together than non-identical. It hints that files with identical names but different binaries are named so out of coincidence not purpose. The act of purposely naming files of different binaries the same seems like it could be more of a malicious pattern.

Each file was then grouped into identical binaries and mapped to the number of copies of that binary that have been seen by our collection process. The cumulative distribution function was then calculated which showed that about half of the binary files in our entire collected dataset have only one copy. This means that our collection has seen every binary an average of two times. The rate at which this number changed with respect to the growing size of our dataset would be an interesting piece of information. If this number increases by a near 1 to 1 with respect to our data set that means that the total number of unique benign binary files is close to the set that our current dataset represents. I imagine this growth would be the case since it seems based on the data that the average machine contains a great deal of the same software. The purple line represents the average file system distance of each copy bucket group. This line is not very informative in and of itself, however, when paired with the two following metrics it can yield some additional information.

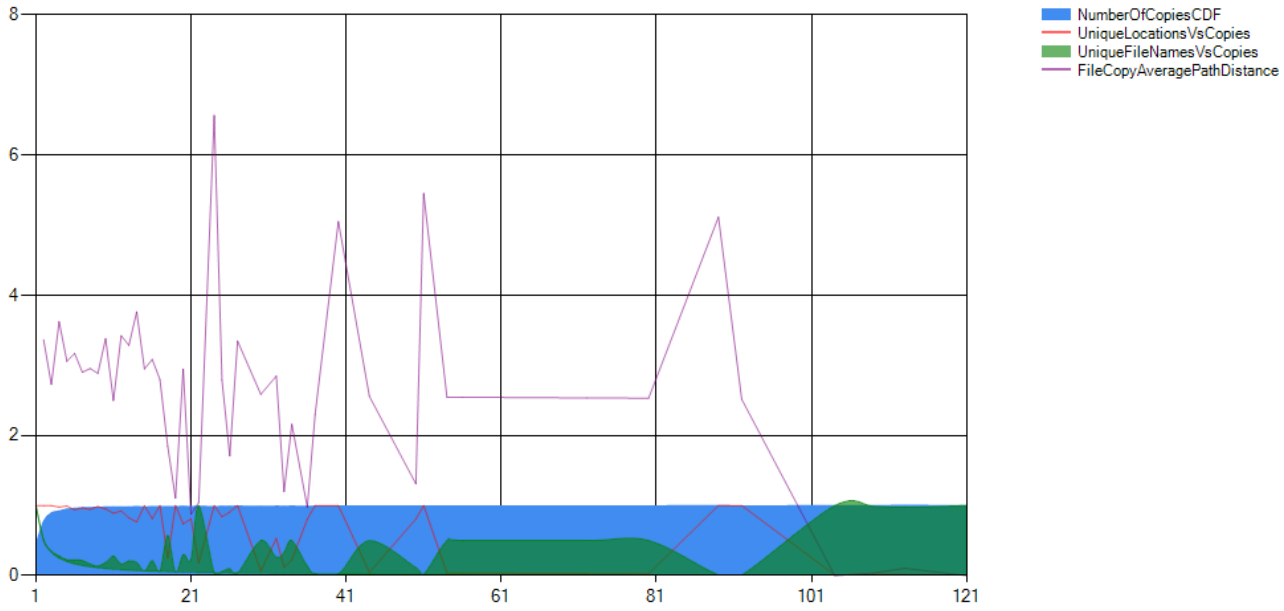


Figure 6 File Binary Graphing

The following metrics distinguish between the three configurations in Figure 7 where files a, b, and c have the same binary (and therefore the same hash) but different names. The common scenario, according to our dataset, is where each identical binary has the same name but are simply in different directories. The less common scenario is where the binaries are all in the same directory but have different names. The least common scenario is where the binaries all have different names and are in different directories. The red line within the blue shows the ratio of unique locations verses copies of the file. If this is close to one, then the scenario is pointed towards the common and uncommon outcomes. If it is close to zero, then it is closer to the less common scenario. The green area is not location based but name and file based. It is similar to the red ratio in the name grouped chart except this only considers files of the same binary. It is the ratio of unique files names to copies. When it is close to 1, the less and uncommon situation arise. When it is close to 1/n the common situation occurs. The least the ratio can be is 1/n where n is the number of copies of the binary. The intersection of the red and green options tell the observer immediately which situation is most strongly occurring in that category. This type of behavior seems to correlate very highly with our dataset. It is because of this that I think this will be one of the best indicators of benign data.

Common	Less	UnCommon
Directory1\a	Directory1\a	Directory1\a
Directory2\a	Directory1\b	Directory2\b
Directory3\a	Directory1\c	Directory3\c

Figure 7 File System Situations

The last grouping is by binary commonality, the ranking of the number of times we saw a file during collections. This shows what type of computer was scanned. Compared to all the files we have seen,

how common are your files? Computers with a greater portion of unique files tend to be machines with development software or other unique tools. Figure 8 below is an example of the commonality difference between our collected dataset (blue) and one of the OIT lab computers (red) that is kept extremely standardized (The x-axis is the uniqueness rank of the number of times seen bin). This shows that the vast majority of software on the OIT lab computer is software that is often found the other computers we scanned. This is one metric we found that is a quick but useful look into what their computer is actually like.

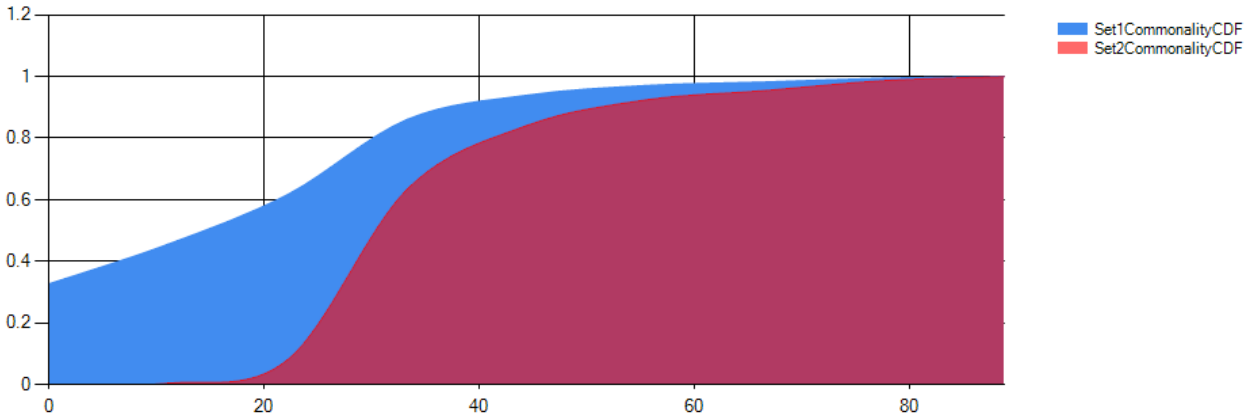


Figure 8 Complete Dataset vs. OIT Lab Machine

In contrast to the data in Figure 8, comparison of an OIT lab computer and an OIT classroom computer show curves that are almost perfectly matched, which is a likely indicator that the two machines share have most of their software in common.

The last addition to this project's analysis was the addition of executable type filters. It gives the application an easy look at what benign software looks like for '.exe' files versus '.dll' files or system files. Analysis of files in the "C:\Windows" folder was also made optional. Since the Windows operating system protects most of its files from being written to by an unauthorized process, we can choose to ignore those files to prevent them from diluting the patterns of more data about more-interesting executable.

Interesting Findings

Although we did not have anywhere near enough data to make sweeping claims about the metadata of benign software, we did find several interesting things that illustrate that this kind of analysis does show the characteristics of the software on the computers we scanned.

Duplicate Files. All of the computers we scanned exhibited large numbers of duplicate files, as matched by SHA-1 hash. Examination of the duplicates revealed that most of them were found in the folders:

- C:\Windows\WinSxS\
- C:\Windows\Installer\

These locations are used by the Windows Installer, Windows Update, and Windows Repair mechanisms to store files used in installs and updates. The number of duplicate files in these folders could be a good indication of a system's age measured by the number of Windows Updates that have been applied.

Similar Disk Images. We chose computers to scan with the intention of collecting data from a variety of Operating System versions. We were able to collect at least two of each of: Windows 7 x86, Windows 7 x64, and Windows Server 2012. One of our findings was that there were differences between the Windows System portions of the scan on machines that should look similar. This likely indicates different update levels and/or additional Windows Features installed on each machine. However, the Windows System portions of the scans of OIT lab computers and OIT classroom computers both run Windows 7 Enterprise x86 and have characteristics much more similar than those of the other machines in the plots of files found. This can be explained by OIT's use of disk images and their practice of regularly (automatically) wiping and reimaging their machines.

Unnecessary Redundancy. We also found many other instances of files occurring in multiple locations and with multiple names. The most interesting of these were files that existed on the same computer with both ".exe" and ".dll" names. We found this most-commonly with files from Adobe products, but also with some Microsoft files (mostly on Windows 8 and Server 2012). We are unsure why a software developer would need to release two identical executable files in the same folder with different file extensions, except to keep to a convention that only files named ".dll" (or ".ocx", etc.) can be dynamically linked to by other programs. (This is true despite the fact that these identical EXE and DLL files contain the same executable entry point and the same API entry points. The DLL version can, in fact, be run as an executable using a utility like Microsoft's RunDLL, and the EXE version can likewise be called as a dynamically linked library from another executable.)

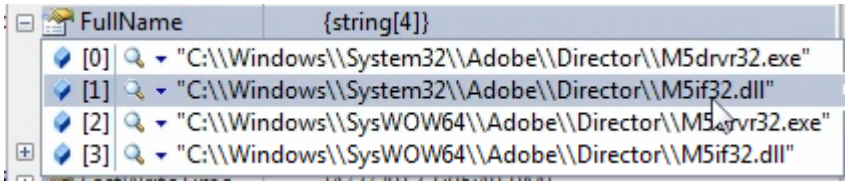


Figure 9 Multiple copies of Adobe Director files[5]

Discussion

Is this worth it?

Our primary focus for this study was not to build an exhaustive corpus of benign software – the sheer amount of benign software out there is far too much for us to handle, notwithstanding that it is out. Our goal was instead to determine, based on what we collected and analyzed, whether we believed it is feasible to build such a corpus (given the resources of a large company with presence on many computers – e.g. Microsoft or and Anti-Malware vendor), and whether we believe that the data in this corpus will be useful. Based on what we have collected and analyzed, we believe that an organization with greater resources and a wider presence than we have should be able to build, maintain, and successfully use this kind of software corpus.

Future Work

Equivalent analysis of bad programs. Our goal was to study the feasibility of creating a corpus containing the metadata of benign software, which could, in the future, be used to determine whether a

[5] Note that there are actually only two files here, not four. The paths containing "System32" are from scanning a 32-bit computer, while the other paths containing "SysWOW64" are the same file on a 64-bit computer.

piece of software is benign by comparing the metadata of that piece of software to the data in the corpus. However, a similar study needs to be made in order to ensure that the metadata of malicious software is not the same as that of benign software. Ideally, the same analysis methods can be applied to a collection of bad software and the results compared to determine what are the differences between the metadata of the two sets. However, even if it turns out that there is no difference between the metadata of good and bad software, this corpus can still be used as part of a whitelist or reputation-based system.

Other Analysis Methods. There may be other analysis methods that would be useful on this dataset, for example, different clustering methods. We used the methods that our group was familiar with, though it is possible that other methods may produce important results.

Automated Analysis. We built a GUI application to analyze our data because we were unsure at the beginning what metrics we would be looking at. Using the GUI was an easy way to add analysis methods and generate graphs. However, in order to be used automatically at full scale, our analysis methods need to be converted into something that can be done in an automated, or at least semi-automated, manner.

More Metadata. We collected the metadata that we could collect, given the resources we had. For greater completeness, future data collection should include additional metadata that we were unable to write code to collect, such as Digital Signatures, file version numbers, and file descriptions. There is also other data that might be useful, such as Compiler or Packer fingerprints, some measure of code obfuscation, and dissemination mechanisms (e.g. physical media, downloaded media, update services, etc). We would also like to look at additional NTFS Alternate Data Streams which we currently do not collect beyond the Zone Identifier (for internet downloads). Because Alternate Data Streams can be entire files by themselves, this is potentially an entire unexplored file structure siting under the visible file structure, in which it would be very easy to hide additional executable files.

Data Error Handling. There are a few inconsistencies in parts of our data, caused by bugs in the SoftwareScanner application and a minor change in database design. While we do not believe that these errors were significant enough to change our results, a larger implementation would need to deal with this issue. However, we did have to throw out the data from most of our earlier scans, which were not collected properly. Another related issue that we did not yet address is path normalization. Collecting data from many computers, some of which are 32-bit and some 64-bit, means that almost every file in “C:\Windows\System32” will also show up in “C:\Windows\SysWOW64”. Normalizing this data so that such files show up only once instead of twice, will be helpful for future analysis. This is also true about executables found in a user’s home directory. Two users with different names who have the same file in their “Downloads” folders will have that file show up as a duplicate. Normalizing user names to “User” in paths beginning with “C:\Users\...” will avoid marking these files as duplicates as well. We also had some issues with the “Strings” utility. Some of the “Unicode Strings” that it gave us for certain files were likely just bits of code that happened to compile to a binary representation that looked like a Unicode string. We need to work on a way to filter those out of the results. Also, there were times when the strings utility threw an error and just printed its own “Usage” text instead of the proper output. In the end, these two issues contributed to making preliminary results from the strings data inconclusive. We also attempted to analyze String data in R, but ran into errors when importing the data and variable size limitations (even in the 64-bit version).

Wider data collection. We collected data on a limited number of computers that we knew we could trust. However, this meant that all of the executables we found were of the type that can be found on the computers of Computer Science college students and their families. Ideally, full-scale data collection will cast a much wider net. The data from the NSRL and/or other databases may be useful as a coverage metric, to see how big the corpus currently is and how far it has left to go.

Issues

SHA-1 Collisions. We chose to use SHA-1 because it is a standard choice for file hashing. There are theoretical attacks on SHA-1 to create deliberate hash collisions, but no actual collisions have been found using this method (Stevens, 2013). We are therefore less worried about purposeful hash collisions, but we do need to look at the possibility of randomly-occurring hash collisions. The probability p of a collision is

$$p \leq \frac{n(n-1)}{2} \times \frac{1}{2^b}$$

where n is the number of files and b is the number of bits in the hash function – for SHA-1, $b = 160$. Assuming that the average computer in our search has $25{,}000$ executables, then $p \leq 2 \times 10^{-40}$, which we feel is sufficient for this purpose.

Deliberate Manipulation. It is possible that, like for any system that relies on differences between trusted and untrusted data, an attacker could develop a program specifically to look like benign software. However, we believe that this is true about all signature-based detection systems and that this is why multiple detection methods are often applied to a single unknown file.

Database Poisoning. Our SoftwareScanner program authenticated to the server by using a shared-secret to calculate an HMAC for the scan results, which was verified on the server. However, because the secret was stored directly in the scanner, this was only secure because we did not release the scanner to the public. Design of a full-scale automated system must balance receiving new data with maintaining the quality of the existing database.

Related Work

There are several other projects that attempt to classify executable files in different ways. Two that we have discussed are Symantec's WINE and Polonium (Chau, Nachenberg, Wilhelm, Wright, & Faloutsos) projects.

Conclusion

We collected and analyzed metadata from benign executables in order to determine whether it is feasible and useful to build a corpus of benign software metadata that can be used to determine whether an unknown file is benign. In this paper, we describe our data collection and analysis methods. We determine that, despite having data only from a small number of computers, there are interesting patterns in the data that mean such a collection may be useful. We also believe that collection of such metadata is doable, and that keeping the corpus up-to-date is feasible for an organization with wide reach, like an Anti-Virus vendor. We also discuss the issues that we had with our collection and analysis and how to mitigate them in a full-scale deployment, and we discuss potential issues that can occur once the system is running at scale.

References

Chau, D. H., Nachenberg, C., Wilhelm, J., Wright, A., & Faloutsos, C. (n.d.). Polonium: Tera-Scale Graph Mining and Inference for Malware Detection. *Proceedings of the second workshop on Large-scale Data Mining: Theory and Applications (LDMTA 2010).* Washington, DC. Retrieved from http://www.cs.cmu.edu/~dchau/polonium_ldmta2010.pdf

Dumitras, T. (2013, September 4). *ENEE 459D | ENEE 759D | CMSC 858Z :: Project Ideas*. Retrieved from ENEE 459D | ENEE 759D | CMSC 858Z :: Fall 2013 :: Security Data Science: http://www.umiacs.umd.edu/~tdumitra/courses/ENEE759D/Fall13/project_ideas.html

Higgins, K. J. (2012, September 4). *McAfee: Close To 100K New Malware Samples Per Day In Q2*. Retrieved from Dark Reading: http://www.darkreading.com/attacks-breaches/mcafee-close-to-100k-new-malware-samples/240006702

NIST. (n.d.). *National Software Reference Library*. Retrieved from http://www.nsrl.nist.gov/

Stevens, M. (2013). *New collision attacks on SHA-1 based on optimal joint local-collision analysis.* CWI, Amsterdam, The Netherlands. Retrieved from http://marc-stevens.nl/research/papers/EC13-S.pdf