Lines of Malicious Code: Insights Into the Malicious Software Industry

Martina Lindorfer
Secure Systems Lab, Vienna
University of Technology
mlindorfer@seclab.tuwien.ac.at

Alessandro Di Federico
NECSTLab, PoliMi, Italy
alessandro.difederico@mail.polimi.it

Federico Maggi
NECSTLab, PoliMi, Italy
fmaggi@elet.polimi.it

Paolo Milani Comparetti
Lastline, Inc.
pmilani@lastline.com

Stefano Zanero
NECSTLab, PoliMi, Italy
zanero@elet.polimi.it

ABSTRACT

Malicious software installed on infected computers is a fundamental component of online crime. Malware development thus plays an essential role in the underground economy of cyber-crime. Malware authors regularly update their software to defeat defenses or to support new or improved criminal business models. A large body of research has focused on detecting malware, defending against it and identifying its functionality. In addition to these goals, however, the analysis of malware can provide a glimpse into the software development industry that develops malicious code.

In this work, we present techniques to observe the evolution of a malware family over time. First, we develop techniques to compare versions of malicious code and quantify their differences. Furthermore, we use behavior observed from dynamic analysis to assign semantics to binary code and to identify functional components within a malware binary. By combining these techniques, we are able to monitor the evolution of a malware's functional components. We implement these techniques in a system we call BEAGLE, and apply it to the observation of 16 malware strains over several months. The results of these experiments provide insight into the effort involved in updating malware code, and show that BEAGLE can identify changes to individual malware components.

Categories and Subject Descriptors

K.6.5 [**Security and Protection**]: Invasive software

General Terms

Security

Keywords

Malware; Evolution; Downloaders; Similarity.

1. INTRODUCTION

ACSAC '12 Dec. 3-7, 2012, Orlando, Florida USA

In parallel with the development of cybercrime into a large underground economy driven by financial gain, malicious software has changed deeply. Originally, malicious software was mostly simple self-propagating code crafted primarily in low-level languages and with limited code reuse. Today, malicious software has turned into an industry that provides the tools that cybercriminals use to run their business [30]. Like legitimate software, malware is equipped with auto-update functionality that allows malware operators to deploy arbitrary code to the infected hosts. New malware versions are frequently developed and deployed; Panda Labs observed 73,000 new malware samples per day in 2011 [5]. Clearly, the majority of these samples are not really new software but rather repacks or incremental updates of previous malware. Malware authors update their code in an endless arms race against security countermeasures such as anti-virus engines and SPAM filters. Furthermore, to succeed in a crowded, competitive market they "innovate", refining the malware to better support cybercriminals' modus operandi or to find new ways to profit at the expense of their victims.

Understanding how malware is updated over time by its authors is thus an interesting and challenging research problem with practical applications. Previous work has focused on constructing the *phylogeny* of malware [15, 20]. Instead, we would like to observe subsequent versions of a malware and automatically characterize its evolution. As a first step, quantifying the differences between versions can provide an indication of the development effort behind this industry, over the observation period.

To provide deeper insight into malicious software and its development, we need to go a step further and identify how the changes between malware versions relate to the functionality of the malware. This is the main challenge that this work addresses. We propose techniques that combine dynamic and static code analysis to identify the component of a malware binary that is responsible for each behavior observed in a malware execution, and to measure the evolution of each component across malware versions.

We selected 16 malware samples from 11 families that included auto-update functionality, and repeatedly ran them in an instrumented sandbox over a period of several months, allowing them to update themselves or download additional components. As a result of dynamic analysis, we obtain the unpacked malware code and a log of its system-level activity. We then compare subsequent malware versions to identify code that is shared with previous versions, and code that was added or removed. From the system-level activity we infer high-level behavior such as "downloading and executing a binary" or "harvesting email addresses". Then, we identify the binary code that implements this observed functionality, and track how it changes over time. As a result, we are able to observe not only the

overall evolution of a malware sample, but also the evolution of its individual functional components.

In summary, this paper makes the following contributions:

- We propose techniques for binary code comparison that are effective on malware and can also be used to contrast a single binary against multiple others, such as against all previous versions of a malware or against a dataset of benign code.
- We propose techniques that use behavior observed from dynamic analysis to assign semantics to binary code. With these techniques, we can identify functional components in a malware sample and track their evolution across malware versions.
- We implement these techniques in a tool called BEAGLE, and use it to automatically monitor 16 malware samples from 11 families over several months and track their evolution in terms of code and expressed behavior. Our results, based on over one thousand executions of 381 distinct malware instances, provide insight into how malware evolves over time and demonstrate BEAGLE's ability to identify changes to malware components.

2. EVOLUTION OF MALICIOUS CODE

Malware is the underlying platform for online crime: From SPAM to identity theft, denial of service attacks or fake-antivirus scams [4], malicious software surreptitiously installed on a victim's computer plays an essential role in criminals' online operations. One key reason for the enduring success of malicious software is the way it has adapted to remain one step ahead of defenses.

Cybercriminals are under constant pressure, both from the security industry, and—as in any market with low barriers to entry—from competing with other criminals. As a result, malicious software is constantly updated to adapt to this changing environment.

The most obvious form of adversarial pressure that the security industry puts on malware authors comes from antivirus (AV) engines: Being detected by widely-deployed AVs greatly reduces the pool of potential victims of a malware. Thus, malware authors strive to defeat AV detection by using packers [29]—also known as "crypters". AV companies respond by detecting the unpacking code of known crypters. The result is that a part of the malicious software industry has specialized in developing crypters that are not yet detected by AVs: Interestingly, Russian underground forums advertise job openings for crypter developers with monthly paychecks of 2,000 to 5,000 US Dollars [21].

This is however only one aspect of the ongoing arms race. SPAM bots try to defeat SPAM filters using sophisticated template-based SPAM engines [34]. Meanwhile, botnets' command and control (C&C) infrastructure is threatened by take downs, so malware authors experiment with different strategies for reliably communicating with their bots [35]. Similarly, the security measures deployed at banking websites and other online services, such as two-factor authentication, require additional malicious development effort. For this, malware authors embed code into the victim's browser that is targeted at a specific website, for instance to mislead him into sending money to a different account than the one he intended to. A plugin implementing such a "web inject" against a specific website for a popular bot toolkit can be worth 2,000 US Dollars [21].

The need to remain one step ahead of defenses is only one reason for malware's constant evolution. Cybercriminals strive to increase their profits—which are threatened by trends such as the declining prices of stolen credentials [22]—by developing new business models. These often require new or improved malware features. As an example, one sample in our dataset implements functionality to simulate a system malfunction on the infected host—presumably to try to sell the victim fake AV or utility software that will "solve" the problem, and another one is able to steal from Bitcoin wallets.

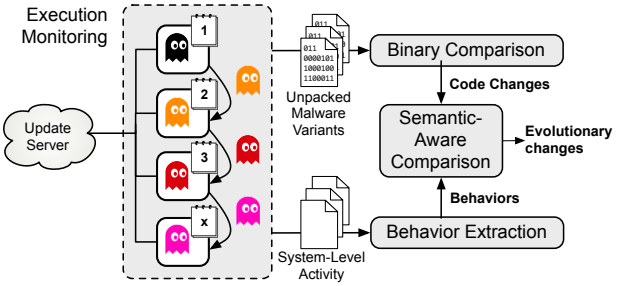


Figure 1: Overview of BEAGLE.

From an economic perspective, we would like to know how expensive it is for a malware author to develop a new feature, as well as how much effort he needs to invest into defeating a security countermeasure. In the absence of direct intelligence on a malware's developers practices, the malware code itself is the richest available source of information on the malware development process. Human analysts routinely analyze malware binaries to understand not only what they can do but also to get a glimpse into the underground economy of cybercrime. In this work, we develop techniques to automate one aspect of this analysis: Measuring how malware changes over time and how these changes relate to the functionality it implements.

3. APPROACH

We implement the approach described in this section in a system we called BEAGLE[1]. Fig. 1 shows an overview of our approach.

BEAGLE first dynamically analyzes a malware sample by running it in an instrumented sandbox, as detailed in §4.1. We allow the sample to connect to its C&C infrastructure to obtain an updated binary. This way, we capture new versions of the analyzed malware. Later, we analyze these updated samples by also executing them in our analysis sandbox. In addition to recording the sample's system- and network-level behavior, the sandbox also acts as a simple generic unpacker [32], and records the unpacked version of the malware binary and of any other executables that are run on the system. This includes malicious code that is injected into benign processes. This allows us to analyze the deobfuscated binary code and compare it across versions with static code-analysis techniques.

Then, BEAGLE compares subsequent versions of a malware's code to quantify the overall evolution of the malware code. Our approach to binary code comparison is based on the control-flow graph's (CFG) structural features, as described in §4.2. This first comparison provides us with an overall measure of how much the code has changed and how much new code has been added, but it offers no insight into the *meaning* of the observed changes.

To attach semantics to the observed changes, we take advantage of the behavioral information provided by our analysis sandbox. The sandbox provides us with a trace of low-level operations performed by the malware. In the behavior extraction phase, discussed in §4.3, we analyze this system-level activity to detect higher-level behaviors such as "sending SPAM", "downloading and executing a binary from the Internet". These behaviors are detected based on a set of high-level rules. In addition to these behaviors that have well-defined semantics (as determined by the human analyst who wrote the detection rule), we also detect behaviors with a-priori unknown semantics, by grouping related system-level events into

[1]HMS Beagle was the ship that Charles Darwin sailed on in the voyage that would provide the opportunity for many of the observations leading to his development of the theory of evolution.

unlabeled behaviors. Once a behavior has been detected, we employ a lightweight technique to map the observed behavior to the code that is responsible for it, and tag individual functions with the set of behaviors they are involved in.

Last, we perform semantic-aware binary comparison as described in §4.4. For this, we combine the results of the behavior-detection step with our binary-comparison techniques to analyze how the code's evolution relates to the observed behaviors. Thus, we are able to monitor when each behavior appears in a malware's codebase and to quantify the frequency and extent of changes to its implementation. The goal is to provide insight into the semantics of the code changes and help the analyst to answer questions such as "What is the overall amount of development effort behind a malware family?", "Within a family, which components are most actively developed?" or "How frequently is a specific component tweaked?".

4. SYSTEM DESCRIPTION

BEAGLE has four modules. The Execution Monitoring module is a sandbox that logs the relevant actions that each sample performs while running (e.g., call stack, system calls, sockets opened, system registry modifications, file creation). The Behavior Extraction module analyzes these logs for each malware sample and extracts high-level behaviors using a set of rules and heuristics. In parallel, the Binary Comparison module disassembles the unpacked binary code. Then, it analyzes the code of each malware sample to find added, removed and shared portions of the CFG across samples, and "labels" these portions with the relative high-level behavior extracted at runtime. The Semantic-Aware Comparison module monitors how the labeled code evolves over time.

4.1 Execution Monitoring

The first step is to obtain subsequent variants of that malware family. One approach would be to rely on the labeling of malware samples by AV engines, and select samples from a specific family as they become available over time. However, this approach has two problems. First, AV engines are not very good at classifying malware largely because their focus is on detection rather than classification [6,25]. Second, different samples of a malware family may be independent forks of a common code base, which are developed and operated by different actors.

The approach we use in BEAGLE is to take advantage of the auto-update functionality included in most modern malware. Specifically, by running malware in a controlled environment and letting it update itself, we can be confident that the intermediate samples that we obtain represent the evolution over time of a malware family—as deployed by a specific botmaster.

Stateful Sandbox. A malware analysis sandbox typically runs each binary it analyzes in a "clean" environment: Before each execution, the state of the system in the sandbox is reset to a snapshot. This ensures that the sample's behavior and other analysis artifacts are not affected by previously-analyzed samples. To allow samples to install and update themselves, however, we need a different approach. A simple yet extremely inefficient solution would be to let the malware run in the sandbox for the entire duration of our experiments. Running a malware sample for months also increases the risk that, despite containment measures, it may cause harm. Instead, we rely on malware's ability to persist on an infected system. Each sample is allowed to run for only a few minutes. At the end of each analysis run, BEAGLE captures the state of the system, in the form of a patch that contains file-system and registry changes to the "clean" system state. Additionally, we detect a malware sample's persistence mechanism by monitoring known auto-start locations [36]. When we want to re-analyze the sample, we start by applying the patch to the "clean" snapshot. Then, we trigger the detected persistence mechanism, so that we effectively simulate a reboot, thus continuing the analysis with updated versions. Hence, in the first execution we observe the installation and update functionality of the original malware sample. In the following executions we observe the updated variants and additional updates.

Generic Unpacking. BEAGLE's analysis sandbox also captures the deobfuscated binary code by acting as a simple, generic unpacker. In its current implementation, BEAGLE simply captures and dumps all code found in memory at process exit or at the end of the analysis run. We do not restrict this to the initial malware process, but also include all code from binaries started by the malware or of processes in which the malware has injected code. For instance, one of the ZeuS samples in our dataset downloads code from a C&C server and injects it into a the `explorer.exe` process. In such a case, BEAGLE also dumps an image of the `explorer.exe` process that includes the injected code.

This simple approach to unpacking, although sufficient for the purpose of our experiments, could be defeated by more advanced packing approaches where unpacked code is re-packed or deleted immediately after use. To address this limitation, BEAGLE could be extended to incorporate more advanced generic unpacking techniques [26, 32].

Sandbox Implementation. BEAGLE's sandbox is an extension of Anubis [8], a dynamic malware analysis sandbox based on the whole-system emulator Qemu. Anubis captures a malware's behavior in the sandbox at the API level, producing a log of the invoked system and API calls, including parameters. Furthermore, Anubis uses dynamic taint tracing to capture data flow dependencies between these calls [9]. In order to facilitate attribution of behavior observed in the sandbox to the code responsible for it, we extended the sandbox to log the call stack corresponding to each API call.

4.2 Binary Comparison

The next component of BEAGLE compares binaries and identifies the code that is shared between versions, the code that was added, or removed. This allows us to quantify the evolution of malicious code by measuring the size of code changes and computing a similarity score between malware variants. More importantly, as discussed in §4.4, we combine this information with code semantics inferred from dynamic behavior to gain an understanding of how evolution relates to malware functionality.

Code Fingerprints. Our technique for binary comparison relies on the structure of the intra-procedural CFG, extending work from [23]. The CFG is a directed graph where nodes are basic blocks and an edge between two nodes indicates a possible control flow transfer (e.g., a conditional jump) between those basic blocks. Following [23], nodes in the CFG are colored based on the classes of instructions that are present in each node, and the problem of finding shared code between two binaries is reduced to searching for isomorphic k-node subgraphs (we use $k = 10$ following [23]). As this problem is intractable, [23] proposed an efficient approximation that relies on extracting a subset of a CFG's k-node connected subgraphs and normalizing them. Each normalized subgraph is a concise representation of a code fragment. In practice, the normalized subgraph is hashed to generate a succinct fingerprint. By matching fingerprints generated from different code samples, we are able to efficiently detect similar code.

As shown by Kruegel et al. [23], these fingerprints are to some extent resistant to code metamorphism, and are effective for detecting code reuse in malware binaries [28]. Here, we take this a step forward and use them to locate the shared code between successive

malware variants. For this, given an unpacked malware binary M, we disassemble it, extract the colored CFG and compute the corresponding set of CFG fingerprints $\mathcal{F}_M$. For each fingerprint f, we also keep track of the addresses $b_M(f)$ of the set of basic blocks it was generated from. For a set of fingerprints $\mathcal{F}$, we indicate the corresponding basic blocks in sample M with $\mathcal{B}_M(\mathcal{F}) = \bigcup_{f \in \mathcal{F}} b_M(f)$. To compare two samples M and N, we compute the intersection of the corresponding fingerprints $I_{M,N} = \mathcal{F}_M \cap \mathcal{F}_N$. The basic blocks corresponding to the fingerprints in this set in either sample correspond to code that is common to M and N. We call them *shared* basic blocks. In sample M, the set of basic blocks that is shared with N is thus $\mathcal{S}_M(M,N) = \mathcal{B}_M(I_{M,N})$. If M and N are two successive variants of a malware $\mathcal{A}_N(M,N) = \mathcal{B}_N \setminus \mathcal{S}_N(M,N)$ (short, $\mathcal{A}(M,N)$) is the set of basic blocks *added* in the new sample N, whereas $\mathcal{R}_M(M,N) = \mathcal{B}_M \setminus \mathcal{S}_M(M,N)$ (short, $\mathcal{R}(M,N)$) is the set of basic blocks *removed* from the old sample M.

Code Whitelisting. The unpacked code produced by our sandbox may include code unrelated to the malware. One reason is that malware may inject code into legitimate processes, that will then be included into our analysis. Furthermore, a malware process sometimes loads system dynamically linked libraries (DLLs) directly into its address space, without using the standard API functions for loading DLLs. In both cases, the unpacked binaries produced by our analysis sandbox will include code that is not part of the malware.

To exclude this code from analysis, we rely on a whitelisting approach. For this, we identify all code (executables and DLLs) in the "clean" sandbox, and compute the set $\mathcal{W}$ of all CFG fingerprints found in this code. In our analysis of each malware sample M, we then identify the basic blocks $\mathcal{B}(\mathcal{W})$ that match these fingerprints. We call them *whitelisted* basic blocks, and do not take them into account for further analysis. An additional benefit of whitelisting code from a clean system is that this can also eliminate code from standard system libraries that has been statically linked into a malware binary.

Similarity and Evolution. We compute the similarity between two malware samples with a variant of the Jaccard set similarity:

$$J(M,N) := \frac{|\mathcal{S}^\star(M,N)|}{|\mathcal{S}^\star(M,N)| + |\mathcal{A}(M,N)| + |\mathcal{R}(M,N)|} \qquad (1)$$

This is roughly the number of shared basic blocks over the total of shared, added and removed basic blocks. The number of shared basic blocks in the two samples $|\mathcal{S}_N(M,N)|$ and $|\mathcal{S}_M(M,N)|$ may differ slightly, because multiple identical k-node subgraphs may exist in a single sample. We mitigate this by picking $|\mathcal{S}^\star(M,N)|$, which is the maximum between the two.

In addition to comparing pairs of samples, we would like to contrast a new malware sample against all previously observed variants, and identify the code that is new in the latest variant. Measuring this code provides the most direct measure of the malware authors' development effort for this variant. For this, we compute the set $\mathcal{F}_{M_1,..,M_{t-1}}$ of fingerprints found in the first $t-1$ samples, and identify the *new basic blocks* $\mathcal{N}_{M_t} = \mathcal{B}_{M_t} \setminus \mathcal{B}_{M_t}(\mathcal{F}_{M_1,..,M_{t-1}})$.

4.3 Behavior Extraction

Automatically understanding the purpose and semantics of binary code is a challenging task. The system-level behavior of a malware sample in an analysis sandbox, however, can be more readily interpreted. To detect specific patterns of malicious behavior previous work [16, 27] has started from system-level events, enriched with data flow information. Martignoni et al. in [27] frame this as a "semantic gap" problem, and propose a technique to detect high-level behavior from system-level events using a hierarchy of manually crafted rules. In the absence of prior knowledge of a pattern of malicious behavior, on the other hand, no such rules are available. As an alternative to leveraging prior knowledge, researchers have used data flow to link individual system-level events into graphs, and applied data mining techniques to a corpus of such graphs to learn to detect malware [11] or to identify its C&C communication [17].

We aim to make our observation of the evolution of malware functionality as complete and insightful as possible. Therefore, we assign semantics to observed behavior that matches known patterns, but also take into account behaviors for which no high-level meaning can be automatically established. In this work, we call the former *labeled*, and the latter *unlabeled* behavior.

Unlabeled Behavior. An unlabeled behavior is a connected graph of system-level events observed during the execution of a sample. Nodes in this graph represent system or API calls, whereas the edges represent data flow dependencies between them. As data flow dependencies are lost when files are written to disk, we also connect nodes that operate on the same file system resources.

Our purpose in taking unlabeled behavior into account is to identify components of the malware and measure their evolution, even though we do not (yet) know what functionality they implement.

Labeled Behavior. A labeled behavior consists of one or more unlabeled behaviors—or, in other words, connected subgraphs of system-level events—such that they match a manually-crafted specification of a known malware behavior. These specifications can take into account the API calls involved, their arguments, as well as the data flow between them. For instance, we define the `DOWNLOAD_EXECUTE` behavior as any data flow dependency from the network to a file that is later executed; another example is the `AUTO_START` behavior, which we define as a write to any of a set of known autostart locations.

Behaviors that remain unlabeled can be examined by an analyst who can assign semantics to them and define behavior specifications for detecting them. Thus, BEAGLE's knowledge-base of behavior specifications grows over time to cover a broader spectrum of malware functionality. As we show in §5.6, a relatively small number of manually-written behavior specifications was sufficient to cover a significant fraction of the malware code that was executed during our experiments.

4.4 Semantic-Aware Comparison

The goal of this step is to attach meaning to the overall changes in the binary code. Essentially, we divide the malware program into a number of functional components. To this end, BEAGLE starts from the behaviors observed at runtime, as discussed in §4.3, and identifies the binary code that is responsible for each observed behavior. By tracking the evolution of this code across malware versions, we are able to measure the evolution of malware's functional components.

Mapping Behavior to Code. The output of the behavior extraction phase is, for each behavior observed in a sample's execution, a sequence of system or API calls with the corresponding call stack. BEAGLE next tries to identify the code responsible for this behavior. BEAGLE works at function granularity: It identifies the set of functions in the unpacked binary that are involved in that behavior, and "tags" them with the behavior. A single function may be tagged with multiple behaviors (e.g., a utility function that is re-used across different functional components). To identify the functions involved in a behavior, we use static analysis to identify a code path that could have been responsible for the observed sequence of calls (taking into account the corresponding call stacks). To do so, we resolve the path between any two consecutive calls by recursively looking up code references to the target function until we find the source function.

Family Name and Label	Source	1st day	Days	Executions	MD5s	Lifespan
Banload TrojanDownloader:Win32/Banload.ADE	(1)	2012-01-31	87	78	3	2.00/83.00/29.33/37.95
Cycbot Backdoor:Win32/Cycbot.G	(1)	2011-09-15	73	73	69	1.00/73.00/2.04/8.60
Dapato Worm:Win32/Cridex.B	(2)	2012-02-24	65	62	25	1.00/43.00/4.60/8.31
Gamarue Worm:Win32/Gamarue.B	(2)	2012-02-10	78	77	19	1.00/76.00/8.47/16.44
GenericDownloader TrojanDownloader:Win32/Banload.AHC	(1)	2012-01-31	82	79	5	2.00/69.00/16.80/26.16
GenericTrojan Worm:Win32/Vobfus.gen!S	(1)	2012-02-07	76	73	55	1.00/44.00/2.71/6.32
Graftor TrojanDownloader:Win32/Grobim.C	(1)	2012-02-17	37	39	22	1.00/17.00/6.00/5.53
Kelihos TrojanDownloader:Win32/Waledac.C	(2)	2012-03-03	56	38	8	1.00/54.00/21.00/22.88
Llac Worm:Win32/Vobfus.gen!N	(1)	2012-02-07	32	33	82	1.00/10.00/1.49/1.71
OnlineGames Worm:Win32/Taterf.D	(1)	2011-09-02	87	80	47	1.00/38.00/3.94/7.28
ZeuS PWS:Win32/Zbot.gen!AF `1be8884c7210e94fe43edb7edebaf15f`	(3)	2012-02-09	79	78	6	1.00/78.00/26.67/28.70
ZeuS PWS:Win32/Zbot `9926d2c0c44cf0a54b5312638c28dd37`	(3)	2012-02-15	74	73	4	1.00/50.00/18.50/19.63
ZeuS PWS:Win32/Zbot.gen!AF* `c9667edbbcf2c1d23a710bb097cddbcc`	(3)	2012-02-23	66	63	6	1.00/36.00/11.00/13.43
ZeuS PWS:Win32/Zbot.gen!AF* `dbedfd28de176cbd95e1cacdc1287ea8`	(3)	2012-02-09	79	78	4	1.00/78.00/20.25/33.34
ZeuS PWS:Win32/Zbot.gen!AF* `e77797372fbe92aa727cca5df414fc27`	(3)	2012-02-10	79	77	5	1.00/77.00/16.20/30.40
ZeuS PWS:Win32/Zbot.gen!AF* `f579baf33f1c5a09db5b7e3244f3d96f`	(3)	2012-03-03	57	55	11	1.00/30.00/5.64/9.75

Table 1: Dataset. The labels in the first columns are based on Microsoft AV naming convention. The MD5 column is the number of distinct binaries encountered. Lifespan is the duration in days of the interval in which an MD5 was observed (min/max/mean/stdev).

We use the addresses in the call stack as landmarks this path should traverse and in case the dynamic path cannot be resolved statically. We tag all functions in the identified path as part of the behavior.

Working at function granularity is a design decision that trades off some precision in delimiting functional components, to achieve performance compatible with a large-scale experiment. As discussed in §4.1, our modified sandbox logs events at the system API level. Previous work that performed a similar mapping of behavior to code at instruction granularity [28], on the other hand, relied on a sandbox logging each executed basic block.

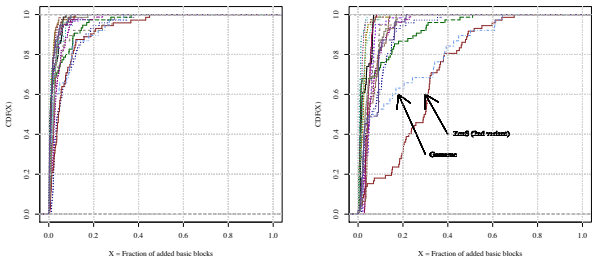
Behavior Evolution. The set of functions that implement a behavior is a functional component of a malware instance. By comparing the components that implement a behavior in successive versions of a malware, we can observe the evolution of that functionality over time. This allows us to get an idea of the development effort involved in updating this functionality by measuring the amount of new code, as well as quickly identifying significant updates to the malicious functionality that may warrant further inspection. For this, we apply the techniques discussed in §4.2 to successive versions of a component, instead of considering entire unpacked binaries. Among the versions of a component observed throughout our experiments, we select the largest implementation by number of basic blocks and call it the *reference behavior*.

Dormant Functionality. Like any dynamic code analysis approach, a limitation of BEAGLE is incomplete code coverage. In a typical execution, a malware sample will reveal only a fraction of the functionality it is capable of: For instance, a bot will send SPAM only if instructed to do so by the botnet's C&C infrastructure. Thus, the techniques described above will identify the presence of each functional component only in some of a sample's executions, even though the code implementing the functionality is present throughout our experiments. This limits our visibility in the component's evolution. In the limit, if a behavior is observed only once, we do not see any evolution. To be able to track evolution in a more complete way, we use the CFG fingerprints from §4.2 to identify a component even in executions where it is *dormant* and the corresponding behavior cannot be observed. For this, we identify the dormant components by locating the functions in a sample that match fingerprints from an *active* (non-dormant) component in another execution.

5. EXPERIMENTAL EVALUATION

5.1 Setup

We run BEAGLE on a desktop-class, dual-core machine with 4GB of RAM, and execute each sample for 15 minutes approximately



(a) $t-1$ vs. t (b) t_0 vs. t

Figure 3: CDF of added basic blocks per family. Day-to-day changes (a) are concentrated around low values for all the families, whereas in the long run (b) each family evolves distinctively, showing different development efforts.

once a day, depending on the workload of the sandbox. Each analysis continues from the state of the previous day's execution in order to analyze only the updated versions. Since we want to observe malware updating itself, we cannot run it in a completely isolated environment, but need to allow it to access the C&C infrastructure from which to obtain updates. To prevent malware from causing harm, we employ containment measures such as redirecting "dangerous" protocols to a local honeypot and limiting bandwidth and connections. These measures cannot guarantee that the malware we run will never cause harm (0-day attacks are especially hard to recognize and block), but we believe that they are sufficient in practice if combined with a prompt response to any abuse complaints (we did not receive any complaints during the course of our experiments).

5.2 Dataset

We selected samples from three different sources: (1) Recent submissions to Anubis for which the data flow detection of Jackstraws [17] indicated download & execute behavior. (2) Malware variants from the top threats according to the Microsoft Malware Protection Center [2] (3) ZeuS samples from ZeuS Tracker [3]. We then discarded samples that showed no update activity in our environment.

As summarized in Tab. 4.3, we analyzed the evolution of 16 samples from 11 families between September 2011 and April 2012. We stopped the analysis and discarded a sample after it failed to contact its C&C server for more than two weeks. Overall, we analyzed a total of 1,023 executions of 381 distinct malware binaries.

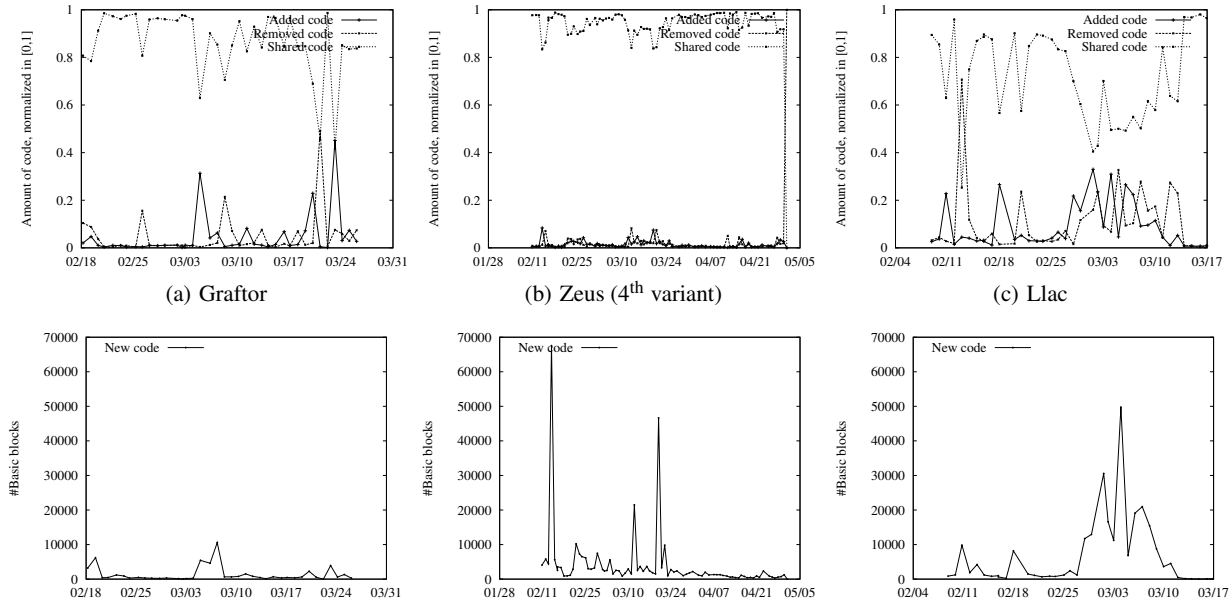


Figure 2: Added, removed and shared code over time. The first row shows the day-to-day changes in the code (i.e., we compare subsequent pairs of samples in the same family), whereas the second row shows the new code. Other families such as Gamarue and GenericDownloader, omitted for space limits, also exhibit interesting evolutions.

5.3 Validation

BEAGLE automatically finds differences between hundreds of malware samples in a few hours on a desktop-class computer. Clearly, an in-depth understanding of the code differences would still require a reverse engineer, which BEAGLE cannot possibly substitute. However, BEAGLE provides valuable guidance to quickly decide what are the interesting changes between malware releases to focus manual inspection. That said, it is hard to assess the correctness of BEAGLE, mainly because we do not have the source code of successive malware versions, and lack a ground truth on the semantics of malware components and their comparison.

The binary comparison component of BEAGLE described in §4.2, however, can be validated by comparing its results against established tools for binary code comparison. For this, we use BinDiff [1, 13], the leading commercial tool for binary comparison and patch analysis, and compare the similarity scores it produces when comparing pairs of samples to those from BEAGLE. Note that BinDiff is based on a completely different approach and uses a number of proprietary heuristics for comparison, and relies on program's call graph, which is not taken into account by our tool. Although the absolute value of BinDiff's similarity score and BEAGLE's similarity score differ, we were able to find a linear transformation[2] from one value to another with a low residual mean square error (6.3% on average, with a peak of 17% for `GenericTrojan`). Note however that BinDiff cannot efficiently be used to contrast a binary against multiple others. BEAGLE's binary comparison component, on the other hand, can be used to contrast a sample against all previous versions (to detect new code), or against a library of benign software (for whitelisting), as discussed in §4.2.

5.4 Overall Changes

In Fig. 2 we show the timelines resulting from comparing samples within three families: Graftor, ZeuS (4th variant), and Llac. We only show three timelines because of space limitations. For the top row of the figure, we consider each consecutive pair of samples of a malware, that is at time t and $t-1$. Then, as described in §4.2, we calculate the amount of added, removed and shared code (Eq. (1)), expressed in distinct basic blocks, between the two samples, and normalized to the total number of distinct basic blocks. For the bottom row, we calculate the absolute amount of code that was never found in any of the previous samples.

These timelines show that overall, day-to-day changes in malware code are relatively small: As would be expected, most new malware

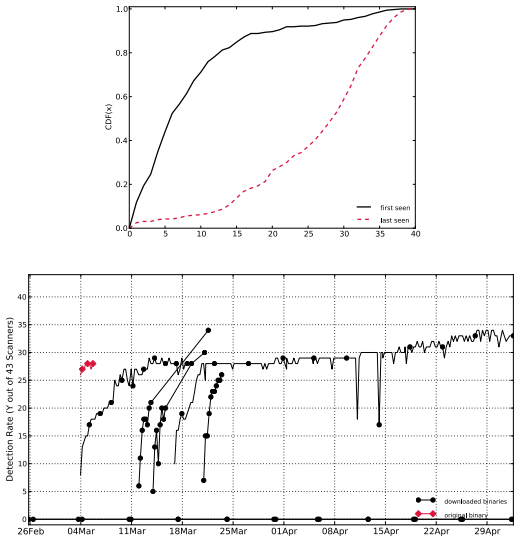


Figure 4: (a) CDF of the detection rate of malware samples among 43 AV engines, at the time each sample is observed for the first and last time in our experiments. (b) Detection rate of Kelihos binaries over time—one line per distinct binary.

[2]To this end, we used `R`'s linear model fitting functions (`lm()` and `poly()`).

FAMILY NAME	%TAGGED	%LABELED	%RATIO	%ADDED	%REMOVED	%SHARED	NEW	#LABELS
Banload	7.31 ± 1.70	6.68 ± 0.75	91.43	2.48 ± 2.96	2.83 ± 3.10	94.69 ± 3.75	176.2 ± 409.2	5
Cycbot	32.36 ± 2.40	31.23 ± 2.95	96.50	10.59 ± 10.36	10.30 ± 10.42	79.11 ± 12.80	1361.4 ± 3937.2	11
Dapato	2.81 ± 1.22	1.15 ± 0.55	40.90	5.15 ± 5.14	5.57 ± 5.63	89.28 ± 7.48	2402.9 ± 7165.3	4
Gamarue	15.90 ± 14.06	14.06 ± 13.40	88.42	12.08 ± 8.16	12.50 ± 9.32	75.41 ± 11.57	2500.1 ± 7747.2	12
GenericDownloader	9.10 ± 1.93	8.58 ± 1.59	94.30	9.80 ± 9.85	9.58 ± 8.81	80.62 ± 12.48	3330.6 ± 7367.8	6
GenericTrojan	22.94 ± 11.05	20.18 ± 10.69	87.97	16.66 ± 16.15	17.03 ± 15.15	66.31 ± 18.76	4974.1 ± 14339.6	11
Graftor	12.66 ± 6.20	9.58 ± 4.70	75.70	6.47 ± 10.40	6.84 ± 9.96	86.69 ± 13.48	682.0 ± 1662.8	4
Kelihos	24.20 ± 2.24	24.09 ± 2.26	99.53	5.18 ± 8.69	5.60 ± 10.10	89.23 ± 12.64	2145.3 ± 4065.3	12
Llac	19.13 ± 14.25	19.11 ± 14.26	99.91	12.82 ± 12.53	14.45 ± 14.70	72.73 ± 19.05	3323.3 ± 7899.1	10
OnlineGames	2.18 ± 0.30	1.96 ± 0.21	89.97	3.35 ± 3.12	3.37 ± 3.12	93.28 ± 5.44	420.0 ± 718.0	9
Zeus	8.37 ± 2.59	6.15 ± 1.32	73.44	2.10 ± 2.24	3.59 ± 11.27	94.31 ± 11.28	1910.8 ± 6148.0	11
Zeus	8.26 ± 1.56	6.44 ± 1.14	78.00	3.65 ± 3.07	5.25 ± 11.85	91.09 ± 12.41	4086.0 ± 11936.3	12
Zeus	10.45 ± 2.67	7.91 ± 2.49	75.73	2.61 ± 2.20	4.51 ± 12.64	92.88 ± 12.47	2234.5 ± 7117.9	11
Zeus	8.55 ± 2.15	6.53 ± 1.19	76.41	2.55 ± 2.51	3.93 ± 11.26	93.52 ± 11.35	2013.6 ± 6874.5	12
Zeus	8.82 ± 1.79	7.73 ± 1.36	87.65	3.12 ± 2.78	4.57 ± 11.33	92.32 ± 11.46	3245.9 ± 7456.3	12
Zeus	7.44 ± 1.31	6.41 ± 0.88	86.06	2.24 ± 2.51	4.53 ± 13.46	93.23 ± 13.46	2523.9 ± 6834.9	13

Table 2: Overall tagged and labeled code (in each version), added, removed, shared code (between consecutive versions), and new code (with respect to all previous versions) for each family (mean±variance, measured in basic blocks). #Labels is the number of distinct behavior labels detected throughout the versions.

versions are incremental updates that reuse most of the code. New code is largely concentrated in a smaller number of peaks, that indicate significant updates to the malware code base. For some families, the amount of brand new code in some of these peaks is significant, up to for instance 50,000 new basic blocks for `Llac`.

Fig. 3 shows a CDF of the similarity of day-to-day differences between successive malware versions (3(a)) and between each version and the first analyzed version (3(b)). Comparing the two graphs clearly shows that while daily updates mostly consist of small changes, for some families the cumulative effect of these small changes eventually result in binaries that are very different from the original sample. This long term evolution varies a lot across the families in our dataset. In Fig.3(b) we highlight `Zeus` (2nd variant) and `Gamarue` that show particularly large cumulative changes.

Tab. 2 summarizes our results, which confirm that, in the majority of cases, the malware writers reuse a significant amount of code when they release day-to-day updates. Remarkably, for a few families the new code added in day-to-day changes accounts for around 10% of the entire malware code on *average*.

5.5 AV Detection

In §2 we suggested that AV engines are one of the main reasons malware authors frequently update their code. During the course of our experiments, we scanned all observed binaries with 43 Anti-Virus engines by using the VirusTotal service. Fig. 4a shows that, as expected, the detection rate of AV engines on a set of binaries—in this case, all binaries executed in our experiments—is generally lower at the beginning of their life cycle than towards the end. More interestingly, we found that in some families, such as Kelihos, as shown in Fig. 4b, the malware writers seem to release a new binary as soon as previous binaries get detected by AV engines. Indeed, whenever a larger number of AV engines start detecting a sample, the malware writers unleash a new, unknown binary, causing a sudden drop in the detection rate.

5.6 Code Behavior

As described in §4.3, we specify behavior as graphs of API calls, connected by data flow, and can take into account API call parameters. Depending on the granularity at which BEAGLE should track changes, an analysts can label behavior by rules that comprise only one function such as `RegSetValue(HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Run,*)` for (one variant of) the `AUTO_START` behavior, or more complex rules such as `InternetOpenUrl|connect -> InternetReadFile|recv -> WriteFile -> WinExec|CreateProcess` for the `DOWNLOAD_EXECUTE` behavior.

Taking into account a-priori knowledge about the behavior of the samples in our dataset and observations during the analysis, we defined rules that label a total of 31 distinct behaviors.

Install. We detect modifications to known autostart locations (`AUTO_START`) to recognize when a sample installs on the system.

Networking. We label network behavior based on protocols and port numbers. We label any data flow to the network over port 25 as `SPAM`. Furthermore, we distinguish between `HTTP_REQUEST`, `HTTPS_REQUEST`, `DNS_QUERY`, general `TCP_TRAFFIC` and `UDP_TRAFFIC` as well `LOCAL_CONNECTION` from and to localhost. Finally, `OPEN_PORT` indicates that the malware listens on a local port.

Download & Execute. This describes the updating functionality of a malware sample by labeling any data flow from the network to files that are then executed (`DOWNLOAD_EXECUTE`) or to memory sections that are injected into foreign processes (`DOWNLOAD_INJECT`).

Information Stealing. We currently detect four information-stealing behaviors. `FTP_CREDENTIALS` and `EMAIL_HARVESTING` indicate that the malware harvests FTP credentials and email addresses, respectively, from the filesystem and registry. `BITCOIN_WALLET` indicates that the malware searches for a Bitcoin wallet to steal digital currency. Finally, `INTERNET_HISTORY` is detected when the malware accesses the user's browsing history.

Fake AV. Fake AV software modifies system settings to simulate system instability and persuade the victim to pay for additional software that fixes these "problems". We detect this as modifications to the registry that disable the task manager (`DISABLE_TASKMGR`) or hide items in the start menu (`HIDE_STARTMENU`), as well as enumeration of the file system and setting all file attributes to hidden (`HIDE_FILES`).

Browser Hijacking. We detect this type of behavior when a malware changes Internet Explorer's or Firefox's proxy settings, (`IE_PROXY_SETTINGS` and `FIREFOX_SETTINGS`).

Anti AV. We detect behavior that disables system call hooks commonly used by AV software by restoring the System Service Dispatch Table rom the disk image of the Windows kernel (`RESTORE_SSDT`).

AutoRun. We detect the use of the `AUTO_RUN` feature as modifications to `autorun.inf`, changes to the AutoRun settings in the registry and the enumeration and spreading to external drives.

Lowering Security Settings. We currently detect three types of behaviors that lower a system's security settings: the creation of new Windows firewall's rules or attempts to disable it completely (FIREWALL_SETTINGS), registry modifications that disable Internet Explorer's phishing filter (IE_SECURITY_SETTINGS), and changes to a system security policy that classifies executables as low risk file types when downloading them from the Internet or opening email attachments (CHANGE_SECURITY_POLICIES).

Miscellaneous. We also detect a number of simple behaviors that have self-explanatory labels: INJECT_CODE, START_SERVICE, EXECUTE_TEMP_FILE, ENUMERATE_PROCESSES and DOWNLOAD_FILE.

Unpacking. To identify a malware's unpacking code (UNPACKER), we do not rely on behavior rules as we do for other labels. Instead, we assume that all code found in the original malware binary *before unpacking* is part of the unpacking behavior. The reason is that malware authors use packing to hide as much as possible of their software from analysis and detection: Thus, all other functionality is typically found inside the packing layer.

The first column of Tab. 2 shows the percentage of a sample's overall code that is tagged with any behavior (labeled or unlabeled). This is all the code that is responsible for *any* observed behavior, and is a measure of the code coverage of our dynamic analysis. Overall coverage is relatively low, which confirms the difficulty of performing a complete dynamic analysis. The second column show the percentage that is tagged with a labeled behavior; That is, code to which we were able to attribute a high-level purpose. Except for one outlier (Dapato, at 40.9%) the labeled code is on average $73.4\% - 99.91\%$ of the total tagged code. This shows that BEAGLE was able to assign most executed code to a functional component.

5.7 Behavior Evolution

With the techniques discussed in §4.4, BEAGLE is able to monitor the evolution of each of the detected behaviors across successive malware versions. For each functional components that implements a behavior, BEAGLE can produce results similar to those presented for the overall malware code in Table 2 and in Figures 2 and 3. Due to space limitations, we can present here only a small sample of these results. To present the evolution of behaviors, we focus on the similarity (as defined in Eq. (1)) between each version of the behavior and the *reference behavior*. As discussed in Section 4.4, the reference behavior is the largest implementation of a behavior by number of basic blocks, across a malware's versions. While this does not provide a complete picture of the code's evolution, it gives an idea of how each behavior grows towards its largest implementation (i.e., the reference behavior).

Fig. 5 shows the similarity over time of each behavior found in ZeuS (3rd variant) against the respective reference behavior. This shows the contribution of each behavior to the overall changes. Interestingly, in this family as well as in other families, we notice very limited code change overall (first plot). However, the breakdown reveals some significant changes towards the end of the observation window, where behaviors such as TCP_CONNECTION, DOWNLOAD_INJECT and HTTP_REQUEST change their similarity with respect to the reference.

A more compact representation of the behavior "variability" is exemplified in Fig. 5, which shows the boxplot distribution of the similarity of each behavior against the respective reference behavior. In the family under examination, which is Gamarue, behaviors such as DOWNLOAD_EXECUTE, UDP_TRAFFIC, and DOWNLOAD_FILE almost never change, except for some outliers (empty circles). Other behaviors, instead, exhibit more variance, which means that their

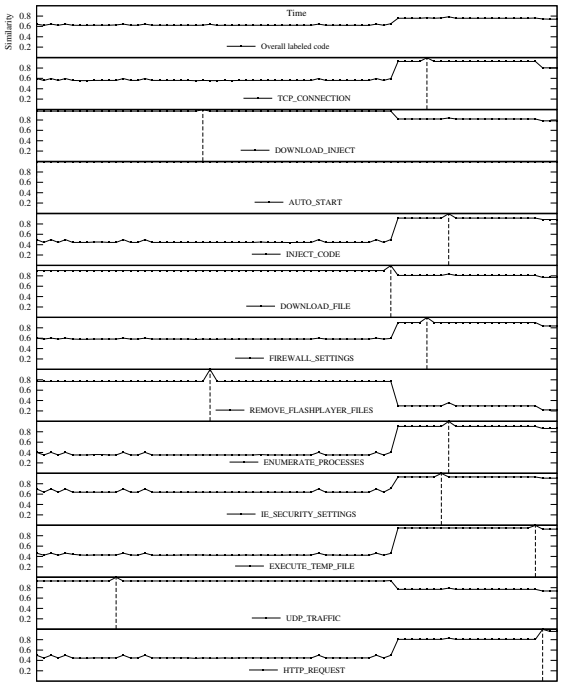


Figure 5: ZeuS (3rd variant): Similarity over time of each behavior against the respective reference behavior. The first timeline is the overall code similarity with respect to the first sample of that family. This plot shows how the overall changes are broke down into changes in the single behaviors. We found analogous patterns also in Gamarue (omitted for space limits).

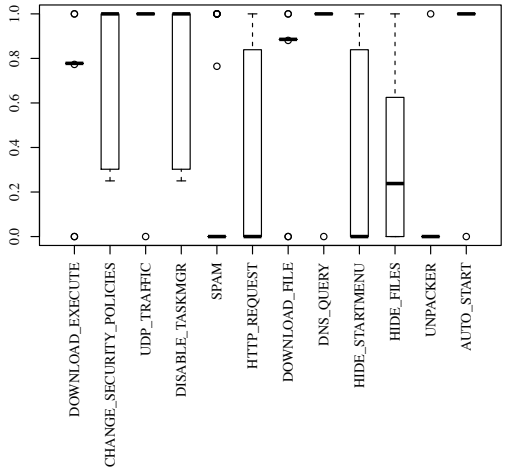


Figure 6: Gamarue: Distribution of the similarity of each behavior against the respective reference behavior. Each box marks the 0%-, 25%- and 75%-, 100%-quantiles, and the median. The circles indicate the outliers.

code is changed often, corresponding to a proportionally larger development effort.

5.8 Lines of Malicious Code

Throughout our evaluation, we have used basic blocks as the unit of measurement for code, whereas it would be more useful to quantify the malware development effort in terms of lines of malicious code. Unfortunately, directly measuring the Lines of

Code (LoC) would require the malware's source code, which is typically unavailable. Nonetheless, we would like to provide a rough estimate of the amount of source code that may correspond to the observed changes. For this, we attempt to estimate a range of possible values for the ratio of basic blocks to LoC in malware samples. Clearly, factors such as the compiler, compiler options and programming paradigms can significantly influence such ratio, thus our estimate is not generalizable outside the scope of our dataset.

We obtained the source code of 150 malware samples of various kinds (e.g., bots, worms, spyware), including the leaked ZeuS source code, and a corresponding executable binary. Apart from ZeuS, none of the families in our dataset are represented in these 150 samples. We then calculated the number of lines of C/C++ source code (LoC), using `cloc`, and the number of basic blocks, using BEAGLE's binary comparison submodule—we excluded the blocks that belong to whitelisted fingerprints. Within this dataset, we found that the ratio between basic blocks and LoC ranges between 50 and 150 blocks per LoC, and around 14.64 for ZeuS. However, one third of the 150 samples that we analyze exhibit a ratio very close to 50 basic blocks per LoC, which we take as a conservative lower bound.

Given these estimates, in the case of ZeuS, the average amount of new code is around 140–280 lines of code, with peaks up to 9,000. Since the Zeus samples in our dataset are closely related to the source code used to estimate this ratio, we consider this a relatively reliable estimate. For the remaining families, with a rough estimate using a ratio of 50, we notice an average amount of *new* code around 100–300 LoC per update cycle, with peaks up to 4,600–9,000. These are just estimates, but they give an overall idea of the significant development effort behind the evolution of malware.

5.9 Conclusive Remarks

BEAGLE revealed that, within our observation window, some families are much more actively developed than others. For instance, GenericTrojan, Llac and ZeuS (3rd variant) have a remarkable amount of new code added. A wider observation window may obviously unveil other "spikes" of development efforts (e.g., new code). As discussed in §5.8, for some families such as ZeuS, we can also give rough estimates of these quantities in lines of source code.

BEAGLE is also able to tell whether and how such changes target each individual behavior. For instance, some behaviors in certain families almost never change (e.g., `UDP_TRAFFIC` or `DOWNLOAD_FILE` in Gamarue), whereas other behaviors (e.g., `HIDE_STARTMENU` or `HTTP_REQUEST` in Gamarue) change over time. In some cases, such as ZeuS (3 rd variant), the overall development effort appears constant, and relatively low. BEAGLE's more focused analysis of the evolution of individual components of this malware, however, reveals that some behaviors undergo significant changes.

6. LIMITATIONS AND FUTURE WORK

Our results demonstrate that BEAGLE is able to provide insight on the real-world evolution of malware samples. However, malware authors could take steps to defeat our system. First of all, the simple unpacking techniques used by BEAGLE could be defeated by using more advanced approaches such as multi-layer or emulation-based packing. For this, BEAGLE could be extended with advanced unpacking approaches [26, 33]. A further problem is that malware analyzed by BEAGLE may be able to detect that it is running in an analysis sandbox, and refuse to update. In recent years, a number of techniques have been proposed to attempt to detect [7, 24] or analyze [12, 19] evasive malware.

Even in the absence of evasion, BEAGLE's dynamic analysis component suffers from limited code coverage. This problem is to some extent alleviated by the fact that we combine information on a malware sample from a large number of executions over a period of months. None-the-less, behavior that is never observed in our sandbox cannot be analyzed. As an example, in our current experiments our observation of the `BITCOIN_WALLET` behavior is incomplete because the analysis environment does not include a Bitcoin wallet for the malware to steal.

BEAGLE can identify and measure the evolution of a malware's functional components. However, it cannot tell us anything more about the semantics of the code changes it detects: This task is currently left to a human analyst. The next logical step is to develop patch analysis techniques to attempt to automatically understand how the update of a component changes its functionality.

7. RELATED WORK

The techniques we apply to measure the similarity between two versions of the same malware family are related to the field of software similarity and classification (as well as plagiarism and theft detection). We refer the reader to [10], which presents a comprehensive review of the existing methods to analyze the similarity of non-binary and binary code, including malicious code.

BinHunt [14] was developed to facilitate patch analysis by accurately identifying which functions have been modified and to what extent. For this, the authors compare the program call graphs and control flow graphs, using symbolic execution and theorem proving to prove that two basic blocks are functionally equivalent, despite transformations such as register substitution and instruction reordering. BinHunt however does not work on packed code, and its efficiency decreases as the amount of code differences increases. BEAGLE's binary comparison component is also based on control flow graphs, and is robust to some code transformations such as basic block and instruction re-ordering and register substitution. While less precise, our approach is fast and scalable and can also be used to contrast a binary against multiple others. Bitshred [18] uses n-grams over binary code and bloom filters to efficiently and scalably locate re-used code in large datasets. Since it relies on raw byte sequences, however, this approach is less robust to syntactic changes in binary code.

The work most closely related to our own is Reanimator [28] which can identify the code responsible for a malware behavior observed in dynamic analysis at instruction granularity and uses CFG fingerprints [23] as signatures to detect the presence of this code in other malware. Our techniques for mapping behavior to code are less precise, and produce results at function granularity. The advantage is that we do not require (extremely slow) instruction-level logging, and are able to apply our techniques to a larger dataset.

Roberts [30] presents some early, high-level insights on the malware development life cycle, largely based on manual malware analysis efforts. More recently, in work concurrent to our own, Rossow et al. [31] performed a large scale analysis of malware downloaders and the C&C infrastructure they rely on.

8. CONCLUSIONS

Understanding the mechanics and economics of malware evolution over time is an interesting and challenging research problem with practical applications. We proposed an automated approach to associate observed behaviors of a malware binary with the components that implement them, and to measure the evolution of each component across malware versions. To the best of our knowledge, no previous research has automatically monitored how malware components change over time.

Our system can observe the overall evolution of a malware sample and of its individual functional components. This led us to interesting insights on the development efforts of malicious code.

Our measurements confirmed commonly held beliefs (e.g., that the malware industry is partly driven by AV advances), but gave also novel and interesting insights. For instance, we observed that most malware writers reuse significant portions of code, but that this varies wildly by family, with significant "spikes" of software development in short timespans. We were also able to distinguish between behaviors that never change in a certain family, and others being constantly developed; In some cases, spikes of development of a malware component are not visible in the overall evolution of the malware sample, but are revealed by the analysis of individual behaviors.

BEAGLE proved to be useful both to build the "big picture" of how and when self-updating malware change, and to guide a malware analysts to the most interesting portions of code (i.e., parts that have changed significantly between two successive versions).

9. ACKNOWLEDGMENTS

The research leading to these results has received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement no 257007.

10. REFERENCES

[1] BinDiff. http://www.zynamics.com/bindiff.html.

[2] Microsoft Malware Protection Center. http://www.microsoft.com/security/portal/Threat/Views.aspx.

[3] ZeuS Tracker. http://zeustracker.abuse.ch/.

[4] Symantec Report on Rogue Security Software. Symantec White Paper, October 2009.

[5] PandaLabs Annual Report. http://press.pandasecurity.com/wp-content/uploads/2012/01/Annual-Report-PandaLabs-2011.pdf, 2011.

[6] M. Bailey, J. Oberheide, J. Andersen, Z. M. Mao, F. Jahanian, and J. Nazario. Automated Classification and Analysis of Internet Malware. In *RAID*, 2007.

[7] D. Balzarotti, M. Cova, C. Karlberger, C. Kruegel, E. Kirda, and G. Vigna. Efficient Detection of Split Personalities in Malware. In *NDSS*, 2010.

[8] U. Bayer, C. Kruegel, and E. Kirda. TTAnalyze: A Tool for Analyzing Malware. In *EICAR Annual Conf.*, 2006.

[9] U. Bayer, P. Milani Comparetti, C. Hlauscheck, C. Kruegel, and E. Kirda. Scalable, Behavior-Based Malware Clustering. In *NDSS*, 2009.

[10] S. Cesare and Y. Xiang. *Software Similarity and Classification*. Springer-Verlag, 2012.

[11] M. Christodorescu, S. Jha, and C. Kruegel. Mining Specifications of Malicious Behavior. In *European Software Engineering Conf. and ACM Symposium on the Foundations of Software Engineering (ESEC-FSE)*, 2007.

[12] A. Dinaburg, P. Royal, M. I. Sharif, and W. Lee. Ether: Malware Analysis via Hardware Virtualization Extensions. In *CCS*, 2008.

[13] H. Flake. Structural Comparison of Executable Objects. In *DIMVA*, 2004.

[14] D. Gao, M. K. Reiter, and D. Song. BinHunt: Automatically Finding Semantic Differences in Binary Programs. In *ICICS*, 2008.

[15] M. Hayes, A. Walenstein, and A. Lakhotia. Evaluation of Malware Phylogeny Modelling Systems Using Automated Variant Generation. *Journal in Computer Virology*, 5(4):335–343, 2009.

[16] G. Jacob, H. Debar, and E. Filiol. Malware Behavioral Detection by Attribute-Automata using Abstraction from Platform and Language. In *RAID*, 2009.

[17] G. Jacob, R. Hund, C. Kruegel, and T. Holz. JACKSTRAWS: Picking Command and Control Connections from Bot Traffic. In *USENIX Security Symposium*, 2011.

[18] J. Jang, D. Brumley, and S. Venkataraman. BitShred: Feature Hashing Malware for Scalable Triage and Semantic Analysis. In *CCS*, 2011.

[19] N. M. Johnson, J. Caballero, K. Z. Chen, S. McCamant, P. Poosankam, D. Reynaud, and D. Song. Differential Slicing: Identifying Causal Execution Differences for Security Applications. In *SSP*, 2011.

[20] M. Karim, A. Walenstein, A. Lakhotia, and L. Parida. Malware Phylogeny Generation Using Permutations of Code. *Journal in Computer Virology*, 1(1):13–23, 2005.

[21] B. Krebs. Criminal Classifieds: Malware Writers Wanted. https://krebsonsecurity.com/2011/06/criminal-classifieds-malware-writers-wanted/, June 2011.

[22] B. Krebs. Banking on Badb in the Underweb. https://krebsonsecurity.com/2012/03/banking-on-badb-in-the-underweb/, Mar. 2012.

[23] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Polymorphic Worm Detection Using Structural Information of Executables. In *RAID*, 2005.

[24] M. Lindorfer, C. Kolbitsch, and P. Milani Comparetti. Detecting Environment-Sensitive Malware. In *RAID*, 2011.

[25] F. Maggi, A. Bellini, G. Salvaneschi, and S. Zanero. Finding Non-trivial Malware Naming Inconsistencies. In *ICISS*, 2011.

[26] L. Martignoni, M. Christodorescu, and S. Jha. OmniUnpack: Fast, Generic, and Safe Unpacking of Malware. In *ACSAC*, 2007.

[27] L. Martignoni, E. Stinson, M. Fredrikson, S. Jha, and J. C. Mitchell. A Layered Architecture for Detecting Malicious Behaviors. In *RAID*, 2008.

[28] P. Milani Comparetti, G. Salvaneschi, E. Kirda, C. Kolbitsch, C. Kruegel, and S. Zanero. Identifying Dormant Functionality in Malware Programs. In *SSP*, 2010.

[29] R. Perdisci, A. Lanzi, and W. Lee. Classification of Packed Executables for Accurate Computer Virus Detection. *Pattern Recognition Letters*, 29(14):1941–1946, 2008.

[30] R. Roberts. Malware Development Life Cycle. *Virus Bulletin Conf.*, (October), 2008.

[31] C. Rossow, C. J. Dietrich, and H. Bos. Large-Scale Analysis of Malware Downloaders. In *DIMVA*, 2012.

[32] P. Royal, M. Halpin, D. Dagon, R. Edmonds, and W. Lee. PolyUnpack: Automating the Hidden-Code Extraction of Unpack-Executing Malware. In *ACSAC*, 2006.

[33] M. Sharif, A. Lanzi, J. Giffin, and W. Lee. Automatic Reverse Engineering of Malware Emulators. In *SSP*, 2009.

[34] B. Stone-Gross, T. Holz, G. Stringhini, and G. Vigna. The Underground Economy of Spam: A Botmaster's Perspective of Coordinating Large-Scale Spam Campaigns. In *USENIX LEET*, 2011.

[35] D. Thomas, K. an Nicol. The Koobface Botnet and the Rise of Social Malware. In *Conf. On Malicious and Unwanted Software (MALWARE)*, 2010.

[36] Y.-M. Wang, R. Roussev, C. Verbowski, A. Johnson, M.-W. Wu, Y. Huang, and S.-Y. Kuo. Gatekeeper: Monitoring Auto-Start Extensibility Points (ASEPs) for Spyware Management. In *USENIX LISA*, 2004.