Automatic Generation of String Signatures for Malware Detection

Kent Griffin Scott Schneider Xin Hu Tzi-cker Chiueh

Symantec Research Laboratories

Abstract. Scanning files for signatures is a proven technology, but exponential growth in unique malware programs has caused an explosion in signature database sizes. One solution to this problem is to use *string signatures*, each of which is a contiguous byte sequence that potentially can match many variants of a malware family. However, it is not clear how to automatically generate these string signatures with a sufficiently low false positive rate. Hancock is the first string signature generation system that takes on this challenge on a large scale.
To minimize the false positive rate, Hancock features a scalable model that estimates the occurrence probability of arbitrary byte sequences in goodware programs, a set of library code identification techniques, and diversity-based heuristics that ensure the contexts in which a signature is embedded in containing malware files are similar to one another. With these techniques combined, Hancock is able to automatically generate string signatures with a false positive rate below 0.1%.

Key words: malware signatures, signature generation, Markov model, library function identification, diversity-based heuristics

1 Introduction

Symantec's anti-malware response group receives malware samples submitted by its customers and competitors, analyzes them, and creates signatures that could be used to identify instances of them in the field. The number of unique malware samples that Symantec receives has grown exponentially in the recent years, because malware programs are increasingly customized, targeted, and intentionally restricted in distribution scope. The total number of distinct malware samples that Symantec observed in 2008 exceeds 1 million, which is more than the combined sum of all previous years.

Although less proactive than desired, signature-based malware scanning is still the dominant approach to identifying malware samples in the wild because of its extremely low false positive (FP) rate, i.e., the probability of mistaking a goodware program for a malware program is very low. For example, the FP rate requirement for Symantec's anti-malware signatures is below 0.1%. Most signatures used in existing signature-based malware scanners are *hash* signatures, each of which is the hash of a malware file. Although hash signatures have a low false positive rate, the number of malware samples covered by each hash signature is also low – typically one. As a result, the total size of the hash signature set grows with the exponential growth in the number of unique malware samples. This creates a signature distribution problem for Symantec: How can we distribute these hash-based malware signatures to hundreds of millions of users across the world several dozen times per day in a scalable way?

One possible solution is to replace hash signatures with *string* signatures, each of which corresponds to a short, contiguous byte sequence from a malware binary. Thus, each string signature can cover many malware files. Traditionally, string signatures are created manually because it is difficult to automatically determine which byte sequence in a malware binary is less FP-prone, i.e., unlikely to appear in any goodware program in the world. Even for manually created string signatures, it is generally straightforward for malware authors to evade them, because they typically correspond to easy-to-modify data strings in malware binaries, such as names of malware authors, special pop-up messages, etc.

Hancock is an automatic string signature generation system developed in Symantec Research Labs that automatically generates high-quality string signatures with minimal FPs and maximal malware coverage. i.e. The probability that a Hancock-generated string signature appears in any goodware program should be very, very low. At the same time each Hancock-generated string signature should identify as many malware programs as possible. Thus, although one string signature takes more space than one hash signature, it uses far less space than all of the hash signatures it replaces.

Given a set of malware samples, Hancock is designed to create a minimal set of N-byte sequences, each of which has a sufficiently low false positive rate, that collectively cover as large a portion of the malware set as possible. Based on previous empirical studies, Hancock sets N to 48. It uses three types of heuristics to test a candidate signature's FP rate: probability-based, disassembly-based, and diversity-based. The first two filter candidate signatures extracted from malware files and the last selects good signatures from among these candidates.

Hancock begins by recursively unpacking malware files using Symantec's unpacking engine. It rejects files that are packed and cannot be unpacked, according to this engine, PEiD [1], and entropy analysis, and stores 48-byte sequences from these files in a list of invalid signatures. Hancock does this because signatures produced on packed files are likely to cover the unpacking code. Blacklisting certain packers should only be done explicitly by a human, rather than through automated signature generation.

Hancock then examines every 48-byte code sequence in unpacked malware files. It finds candidate signatures using probability-based and disassembly-based heuristics: it filters out byte sequences whose estimated occurrence probability in goodware programs, according to a pre-computed goodware model, is above a certain threshold; that are considered a part of library functions; or whose assembly instructions are not sufficiently interesting or unique, based on heuristics that encode malware analysts' selection criteria. It examines only code so that disassembly-based heuristics can work and because malware authors can more easily vary data.

Among those candidate signatures that pass the initial filtering step, Hancock further applies a set of selection rules based on the *diversity* principle: If the set of malware samples containing a candidate signature are similar, then they are less FP-prone. A candidate signature in a diverse set of malware files is more likely to be a part of a library used by several malware families. Though identifying several malware families seems like a good idea, if a signature is part of library code, goodware files might use the same library. On the other hand, if the malware files are similar, they are more likely to belong to one family and the candidate signature is more likely to be code that is unique to that family.

Finally, Hancock is extended to generate string signatures that consist of multiple disjoint byte sequences rather than only one contiguous byte sequence. Although multi-component string signatures are more effective than single-component signatures, they also incur higher run-time performance overhead because individual components are more likely to match goodware programs. In the following sections, we will describe the signature filter algorithms, the signature selection algorithms, and the multi-component generalization used in Hancock.

2 Related Work

Modern anti-virus software typically employ a variety of methods to detect malware programs, such as signature-based scanning [2], heuristic-based detection [3], and behavioral detection [4]. Although less proactive, signature-based malware scanning is still the most prevalent approach to identify malware because of its efficiency and low false positive rate. Traditionally, the malware signatures are created manually, which is both slow and error-prone. As a result, efficient generation of malware signatures has become a major challenge for anti-virus companies to handle the exponential growth of unique malware files. To solve this problem, several automatic signature generation approaches have been proposed.

Most previous work focused on creating signatures that are used by Network Intrusion Detection Systems (NIDS) to detect network worms. Singh et al. proposed EarlyBird [5], which used packet content prevalence and address dispersion to automatically generate worm signatures from the invariant portions of worm payloads. Autograph [6] exploited a similar idea to create worm signatures by dividing each suspicious network flow into blocks terminated by some breakmark and then analyzing the prevalence of each content block. The suspicious flows are selected by a port-scanning flow classifier to reduce false positives. Kreibich and Crowcroft developed Honeycomb [7], a system that uses honeypots to gather inherently suspicious traffic and generates signatured by applying the longest common substring (LCS) algorithm to search for similarities in the packet payloads. One potential drawback of signatures generated from previous approaches is that they are all continuous strings and may fail to match polymorphic worm payloads. Polygraph [8] instead searched for invariant content in the network flows and created signatures consisting of multiple disjoint content substrings. Polygraph also utilized a naive Bayes classifier to allow the probabilistic matching and classification, and thus provided better proactive detection capabilities. Li et al. proposed Hasma [9], a system that used a model-based algorithm to analyze the invariant contents of polymorphic worms and analytically prove the attack-resilience of generated signatures. PDAS (Position-Aware Distribution Signatures) [10] took advantage of a statistical anomaly-based approach to improve the resilience of signatures to polymorphic malware variants. Another common method for detecting polymorphic malware is to incorporate semantics-awareness into signatures. For example, Christodorescu et al. proposed static semantics-aware malware detection in [11]. They applied a matching algorithm on the disassembled binaries to find the instruction sequences that match the manually generated templates of malicious behaviors, e.g., decryption loop. Yegneswaran et al. developed Nemean [12], a framework for automatic generation of intrusion signatures from honeynet packet traces. Nemean applied clustering techniques on connections and sessions to create protocol-semantic-aware signatures, thereby reducing the possibility of false alarms.

Another loosely related area is the automatic generation of attack signatures, vulnerability signatures and software patches. TaintCheck [13] and Vigilante [14] applied taint analysis to track the propagation of network inputs to data used in attacks, e.g., jump addresses, format strings and system call arguments, which are used to create signatures for the attacks. Other heuristic-based approaches [15–18] have also been proposed to exploit properties of specific exploits (e.g., buffer overflow) and create attack signatures. Generalizing from these approaches, Brumley et al. proposed a systematic method [19] that used a formal model to reason about vulnerability signatures and quantify the signature qualities. An alternative approach to preventing malware from exploiting vulnerabilities is to apply data patches (e.g. Shield vulnerability signatures [20]) in the firewalls to filter malicious traffic. To automatically generate data patches, Cui et al. proposed ShieldGen [21], which leveraged the knowledge of data format of malicious attacks to generate potential attack instances and then created signatures from the instances that successfully exploit the vulnerabilities.

Hancock differs from previous work by focusing on automatically generating high-coverage string signatures with extremely low false positives. Our research was based loosely on the virus signature extraction work [22] by Kephart and Arnold, which was commercially used by IBM. They used a 5-gram Markov chain model of good software to estimate the probability that a given byte sequence would show up in good software. They tested hand-generated signatures and found that it was quite easy to set a model probability threshold with a zero false positive rate and a modest false negative rate (the fraction of rejected signatures that would not be found in goodware) of 48%. They also generated signatures from assembly code (as Hancock does), rather than data, and identified candidate signatures by running the malware in a test environment. Hancock does not do this, as dynamic analysis is very slow in large-scale applications.

Symantec acquired this technology from IBM in the mid-90s and found that it led to many false positives. The Symantec engineers believed that it worked well for IBM because IBM's anti-virus technology was used mainly in corporate environments, making it much easier for IBM to collect a representative set of goodware. By contrast, signatures generated by Hancock are mainly for home users, who have a much broader set of goodware. The model's training set cannot possibly contain, or even represent, all of this goodware. This poses a significant challenge for Hancock in avoiding FP-prone signatures.

3 System Overview

The Hancock system automatically generates string signatures for detecting malware. When an anti-virus engine scans a file to see if it is malware, it looks for these signatures, which are simple byte sequences. Hancock creates signatures that each cover as many malware files as possible. This system has several benefits:

1. **Smaller signature set** Symantec's signature set is currently tens of megabytes, and growing exponentially. If every signature were to cover many files, that would significantly slow signature set growth. Hancock can do this and also generate replacements for old signatures, reducing the signature set's size.
2. **Saving manpower** Currently, anti-virus signatures are created manually. This not only limits the number of malware files that can be signatured; it also reduces the quality of signatures, since engineers are so pressed for time. Automatic signature generation will allow them to spend more time creating better signatures for the tough cases that Hancock cannot handle.
3. **Coverage and 0-day detection** If each signature covers many sample files in a malware family, then it has a greater chance of detecting new variants in that malware family. This saves more effort for signature-creating engineers and provides 0-day protection against new malware variants.

The Hancock system operates primarily by analyzing a specified set of malware files – typically thousands or tens of thousands. At a high level, its phases are:

1. Training a model using a goodware training set.
2. Unpacking files and identifying files that cannot be unpacked.
3. Disassembling a set of malware files with IDA Pro.
4. Generating a set of candidate signatures from the malware.
5. Applying a series of heuristics to these candidate signatures, favoring those that cover many malware files and weeding out those that are likely to exist in goodware files.

Section 4.1 will describe how the Hancock system trains and uses the model.

Hancock does not directly deal with the issue of file packing. It recursively unpacks malware as well as it can, using Symantec's anti-virus engine and an internal Symantec tool. Then it identifies malware files that are still packed, using PEiD[1], entropy analysis, and internal tools. Hancock generates signatures only on unpacked files.

The system does not create signatures from packed files because, if a packed file obfuscates its malware payload, the only signatures Hancock could generate would be on the unpacker itself. This sort of decision should only be made explicitly, for each packer, rather than by an automated system.

Hancock does not simply throw away packed files. It scans them for byte sequences that could normally be candidate signatures. These sequences are blacklisted as candidate signatures, when Hancock scans through unpacked malware files.

Hancock only draws candidate signatures from code. This is because many of our heuristics only work on assembly code. Also, Symantec response engineers prefer code signatures because they are harder (or, at least, less trivial) to obfuscate. Thus, Hancock unpacks malware files by disassembling them with IDA Pro.

Signatures are 48 bytes long. This signature length trades off false positives with coverage. If it were shorter, Hancock could generate signatures to cover more malware files, but the false positive rate would go up; and vice versa for longer signatures. Empirical testing showed 48 bytes to be a good trade-off.

Hancock generates candidate signatures by examining every code sequence in unpacked malware files. It applies heuristics that only apply to the content of a signature:

1. The model estimates the probability that a candidate signature will be found in goodware. See section 4.1.
2. Hancock rejects any code that IDA Pro's FLIRT (Fast Library Identification and Recognition Technology) identifies as part of a library function. Section 4.2 describes extensions on this.
3. Hancock examines the assembly instructions and instruction operands in the signature. Section 4.3 describes how Hancock judges if a candidate signature is "interesting" enough.

This generates a large set of candidate signatures. In the final step, Hancock examines each signature, starting with those that apply to the most malware files. It applies a second set of heuristics that look at the context of each signature in each containing malware file. If a signature has too much *diversity* – if its use in the containing malware files is too different – then this signature is too generic. Either it is a rare library function that IDA's FLIRT did not identify or just generic.

Some of these diversity-based heuristics are based on the raw bytes of the containing malware files, or just the malware files themselves. Others analyze the assembly instructions in and around the candidate signatures. Section 5 describes them in detail.

Two themes carry through all of these heuristics for weeding out false positives:

– **Discrimination power** None of these heuristics are perfect. All have false positive and false negative rates. They are still useful, however, if they are more likely to reject a bad candidate signature than a good one. This is the heuristic's *discrimination power*. The Hancock system can still get good coverage and a low false positive rate by applying many such heuristics.

– **Independence of heuristics** The discrimination power of several heuristics will only add if the two heuristics are indepedent. If the false positives for two heuristics overlap too much, then they are too redundant to both be useful.

3.1 The FP Check Index

The False Positive Check Index is a tool that very quickly tests if a specified byte sequence exists in a large goodware set. The index consists of a database of short goodware byte sequences and an algorithm for searching for a larger, specified byte sequence. The cost is that this database's size is on the same order as the original data.

An index is created with a fixed stride, S, and byte chunk length, L, with $L \geq S$. On adding a goodware file to the index, the index's database stores every L-byte chunk with a byte offset that is a multiple of S. The database maps each chunk back to the set of files (and file locations) that have that chunk.

When the index searches for a byte sequence, it checks the database for S chunks. These chunks are bytes number 0 to $L-1$, 1 to L, ... $S-1$ to $L+S-2$ from the specified byte sequence. If a chunk exists in the database, the index looks for the whole byte sequence in the corresponding original file locations. One can see that the shortest byte sequence an index can check is $L+S-1$ bytes long.

The index uses the least size when $L = S$ and S is large. Setting $L \geq S$ can speed up the index, but we have found that the index is fast enough. In our testing, our largest index was on 32 GB of goodware. The database used an additional 52 GB of disk space and could check 100 byte sequences per second.

A Symantec product team is in the process of adopting the index (independent of the whole Hancock system) for use. Currently, the test manually created signatures by scanning a goodware set with the full anti-virus engine. This takes several hours. They plan to use the index to get instant feedback on manually created signatures, which will save the time of retesting a corrected signature set from scratch.

4 Signature Candidate Selection

4.1 Goodware Modeling

The first line of defense in Hancock is a Markov chain-based model that is trained on a large goodware set and is designed to estimate the probability of a given byte sequence appearing in goodware. If the probability of a candidate signature appearing in some goodware program is higher than a threshold, Hancock rejects it. Compared with standard Markov models, Hancock's goodware model has two important features:

– **Scalable to very large goodware set** Symantec regularly tests its anti-virus signatures against several terabytes of goodware programs. A standard Markov model uses linear space [23] in the training set size, with a large constant factor. Hancock's goodware model focuses only on high-information-density byte sequences so as to scale to very large goodware training sets.
– **Focusing on rare byte sequences** For a candidate signature to not cause a false positive, its probability of appearing in goodware must be very, very low. Therefore, the primary goal of Hancock's model is to distinguish between low-probability byte sequences and very rare byte sequences.

Basic Algorithm The model used in Hancock is a fixed-order 5-gram Markov chain model, which estimates the probability of the fifth byte conditioned on the occurrence of the preceding four bytes. Training consists of counting instances of 5-grams – 5-byte sequences – as well as 4-grams, 3-grams, etc. The model calculates the probability of a 48-byte sequence by multiplying estimated probabilities of each of the 48 bytes. A single byte's probability is the probability of that byte following the four preceding bytes. For example, the probability that "e" follows "abcd" is

$$p(\mathrm{e}|\mathrm{abcd}) = \frac{count(\mathrm{abcde})}{count(\mathrm{abcd})} * (1 - \epsilon(count(\mathrm{abcd}))) + p(\mathrm{e}|\mathrm{bcd}) * \epsilon(count(\mathrm{abcd}))$$

In this equation, $count(s)$ is the number of occurrences of the byte sequence s in the training set. We limit overtraining with $\epsilon(count(s))$, the *escape mass* of s. Escape mass decreases with count. Empirically, we found that a good escape mass for our model is $\epsilon(c) = \frac{\sqrt{32}}{\sqrt{32}+\sqrt{c}}$.

Model Pruning The memory required for a vanilla fixed-order 5-gram model is significantly greater than the size of the original training set. Hancock reduces the memory requirement of the model by incorporating an algorithm that prunes away less useful grams in the model. The algorithm looks at the *relative information gain* of a gram and eliminates it if its information gain is too low. This allows Hancock to keep the most valuable grams, given a fixed memory constraint.

Consider a model's grams viewed as nodes in a tree. The algorithm considers every node X, corresponding to byte sequence s, whose children (corresponding to $s\sigma$ for some byte σ) are all leaves. Let s' be s with its first byte removed. For example, if s is "abcd", s' is "bcd". For each child of X, σ, the algorithm compares $p(\sigma|s)$ to $p(\sigma|s')$. In this example, the algorithm compares $p(\mathrm{e}|\mathrm{abcd})$ to $p(\mathrm{e}|\mathrm{bcd})$, $p(\mathrm{f}|\mathrm{abcd})$ to $p(\mathrm{f}|\mathrm{bcd})$, etc. If the difference between $p(\sigma|s)$ and $p(\sigma|s')$ is smaller than a threshold, that means that X is does not add that much value to σ's probability and the node σ can be pruned away without compromising the model's accuracy.

To focus on low-probability sequences, Hancock uses the difference between the logs of these two probabilities, rather than that between their raw probability values. Given a space budget, Hancock keeps adjusting the threshold until it hits the space target.

Model Merging Creating a pruned model requires a large amount of intermediate memory, before the pruning step. Thus, the amount of available memory limits the size of the model that can be created. To get around this limit, Hancock creates several smaller models on subsets of the training data, prunes them, and then merges them.

Merging a model M_1 with an existing model M_2 is mostly a matter of adding up their gram counts. The challenge is in dealing with grams pruned from M_1 that exist in M_2 (and vice versa). The merging algorithm must recreate these gram counts in M_1. Let $s\sigma$ be such a gram and let s' be s with its first byte removed. The algorithm estimates the count for $s\sigma$ as $count(s) * p(\sigma|s')$. Once these pruned grams are reconstituted, the algorithm simply adds the two models' gram counts.

Experimental Results We created an occurrence probability model from a 1-GByte training goodware set and computed the probability of a large number of 24-byte test sequences, extracted from malware files. We checked each test byte sequence against a goodware database, which is a large superset of the training set, to determine if it is a true positive (a good signature) or a false positive (which occurs in goodware). In Figure 1, each point in the FP and TP curves represents the fraction (Y axis value) of test byte sequences whose model probability is below the X axis value.

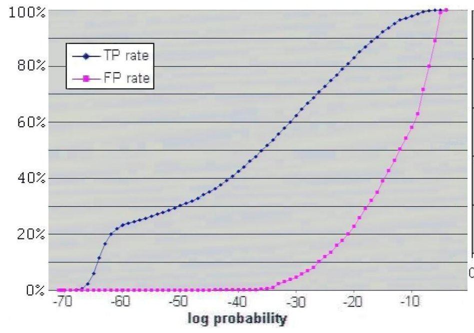


Fig. 1. Fractions of FP and TP sequences with probabilities below the X value

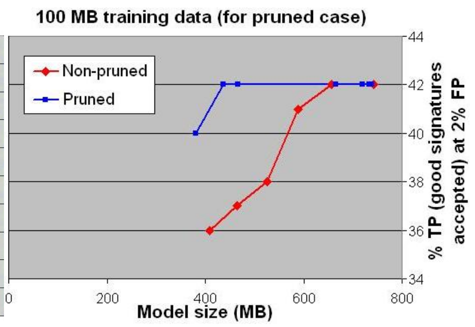


Fig. 2. TP rate comparison between models with varying pruning thresholds and varying training set sizes

As expected, TP signatures have much lower probabilities, on average, than FP signatures. A small number of FP signatures have very low probabilities – below 10^{-60}. Around probability 10^{-40}, however, the model does provide excellent discrimination power, rejecting 99% of FP signatures and accepting almost half of TP signatures.

To evaluate the effectiveness of Hancock's information gain-based pruning algorithm, we used two sets of models: non-pruned and pruned. The former were trained on 50 to 100 Mbytes of goodware. The latter were trained on 100 Mbytes of goodware and pruned to various sizes. For each model, we then computed its TP rate at the probability threshold that yields a 2% FP rate. Figure 2 shows these TP rates of goodware models versus the model's size in memory. In this case, pruning can roughly halve the goodware model size while offering the same TP rate as the pruned model derived from the same training set.

4.2 Library Function Recognition

A library is a collection of standard functions that implement common operations, such as file IO, string manipulation, and graphics. Modern malware authors use library functions extensively to simplify development, just like goodware authors. By construction, variants of a malware family are likely to share some library functions. Because these library functions also have a high probability of appearing in goodware, Hancock needs to remove them from consideration when generating string signatures. Toward this goal, we developed a set of library function recognition techniques to determine whether a function in a malware file is likely to be a library function or not.

A popular library identification technique is IDA Pro's Fast Library Identification and Recognition Technology (FLIRT) [24], which uses byte pattern matching algorithms (similar to string signature scanning) to quickly determine whether a disassembled function matches any of the signatures known to IDA Pro.[1] Although FLIRT is very accurate in pinpointing common library functions, it still needs some improvement to suit Hancock's needs. First, FLIRT is designed to never falsely identify a library. To achieve this, FLIRT first tries to identify the compiler type (e.g., Visual C++ 7.0, 8.0, Borland C++, Delphi, etc.) of a disassembled program and applies only signatures for that compiler. For example, vcseh signatures (Structured Exception Handling library signatures) will only be applied to binary files that appear to have been compiled with Visual C++ 7 or 8. This conservative approach can lead to false negatives (a library function not identified) because of failure in correctly detecting the compiler type. In addition, because FLIRT uses a rigorous pattern matching algorithm to search for signatures, small variation in libraries, e.g., minor changes in the source code, different settings in compiler optimization options or use of different compiler versions to build the library, could prevent FLIRT from recognizing all library functions in a disassembled program.

In contrast to FLIRT's conservative approach, Hancock's primary goal is to eliminate false positive signatures. It takes a more aggressive stance by being willing to mistake non-library functions for library functions. Such misidentification is acceptable because it prevents any byte sequence that is potentially

[1] IDA Pro ships with a database of signatures for about 120 libraries associated with common compilers. Each signature corresponds to a binary pattern in a library function.

associated with a library function from being used as a malware signature. We exploited this additional latitude with the following three heuristics:

Universal FLIRT Heuristic This heuristic generalizes IDA Pro's FLIRT technique by matching a given function against all FLIRT signatures, regardless of whether they are associated with the compiler used to compile the function. This generalization is useful because malware authors often post-process their malware programs to hide or obfuscate compiler information in an attempt to deter any reverse engineering efforts. Moreover, any string signatures extracted from a function in a program compiled by a compiler C1 that looks like a library function in another compiler C2 are likely to cause false positives against programs compiled by C2 and thus should be rejected.

Library Function Reference Heuristic This heuristic identifies a library function if the function is statically called, directly or indirectly, by any known library function. The rationale behind this heuristic is that since a library cannot know in advance which user program it will be linked to, it is impossible for a library function to statically call any user-written function, except callback functions, which are implemented through function pointers and dynamically resolved. As a result, it is safe to mark all children of a library function in its call tree as library functions. Specifically, the proposed technique disassembles a binary program, builds a function call graph representation of the program, and marks any function that is called by a known library function as a library function. This marking process repeats itself until no new library function can be found.

In general, compilers automatically include into an executable binary certain template code, such as startup functions or error handling, which IDA Pro also considers as library functions as well. These template functions and their callees must be excluded in the above library function marking algorithm. For example, the entry point function *start* and *mainCRTstartup* in Visual C++-compiled binaries are created by the compiler to perform startup preparation (e.g., execute global constructors, catch all uncaught exceptions) before invoking the user-defined main function.

Address Space Heuristic This heuristic identifies a library function based on whether its neighboring functions in the binary file are library functions. When a library is statically linked into a program, the binary codes of the library usually occupy a contiguous address space range and sometimes there is padding space between adjacent functions. Therefore, to detect those internal, non-exported functions in a library that we cannot prove statically are called by some known library function, we exploit the physical proximity property of functions that belong to the same library. More specifically, we mark a function as a library functions if:

– It is immediately surrounded or sandwiched by known library functions, and

– The size of the space between it and its neighboring library functions is below a certain threshold. In Hancock, we set the threshold to be 128 bytes, based on a statistical analysis of inter-library-function space in binary programs generated by commercial compilers.

In Hancock, we implement the above library function heuristics as an IDA Pro plugin. After IDA Pro disassembled a malware program, Hancock first applies the Universal FLIRT heuristic to maximize the detection capability with FLIRT's library signatures. Then the Function Reference and the Address Space heuristics are repeated until the set of identified library functions converges. Although these techniques collectively could mistake non-library functions for library functions, in practice such false positives do not have noticeable impacts on the effectiveness of Hancock's overall signature generation capability. However, by eliminating up front more of the shared library code in binaries, we have found that we can significantly reduce the number of false positives signatures. In addition, these techniques also improve the signature quality, because it allows Hancock to focus more on their coverage than on their FP likelihood.

4.3 Code Interestingness Check

The *code interestingness* check is designed to capture the intuitions of Symantec's malware analysis experts about what makes a good string signature. For the most part, these metrics identify signatures that are less likely to be false positives. They can also identify malicious behavior, though avoiding false positives is the main goal. The code interestingness check assigns a score for each "interesting" instruction pattern appearing in a candidate signature, sums up these scores, and rejects the candidate signature if its sum is below a threshold, i.e. not interesting enough. The interesting patterns used in Hancock are:

– **Unusual constant values** Constants sometimes have hard-coded values that are important to malware, such as the IP address and port of a command and control server. More importantly, if a signature has unusual constant values, it is less likely to be a false positive.
– **Unusual address offsets** Access to memory that is more than 32 bytes from the base pointer can indicate access to a large class or structure. If these structures are unique to a malware family, then accesses to particular offsets into this structure are less likely to show up in goodware. This pattern is not uncommon among legitimate Win32 applications. Nonetheless, it has good discrimination power.
– **Local or non-library function calls** A local function call itself is not very distinctive, but the setup for local function calls often is, in terms of how it is used and how its parameters are prepared. In contrast, setup for system calls is not as interesting, because they are used in many programs and invoked in a similar way.
– **Math instructions** A malware analyst at Symantec noted that malware often perform strange mathematical operations, to obfuscate and for various

other reasons. Thus, Hancock looks for strange sequences of XORs, ADDs, etc. that are unlikely to show up in goodware.

5 Signature Candidate Filtering

Hancock selects candidate signatures using techniques that assess a candidate's FP probability based solely on its contents. In this section, we describe a set of filtering techniques that remove from further consideration those candidate signatures that are likely to cause a false positive based on the signatures' use in malware files.

These *diversity-based* techniques only accept a signature if it matches variants of one malware family (or a small number of families). This is because, if a byte sequence exists in many malware families, it is more likely to be library code – code that goodware could also use. Therefore, malware files covered by a Hancock signature should be similar to one another.

Hancock measures the diversity of a set of binary files based on their byte-level and instruction-level representations. The following two subsections describe these two diversity measurement methods.

5.1 Byte-Level Diversity

Given a signature, S, and the set of files it covers, X, Hancock measures the byte-level similarity or diversity among the files in X by extracting the byte-level context surrounding S and computing the similarity among these contexts. More concretely, Hancock employs the following four types of byte-level signature-containing contexts for diversity measurement.

Malware Group Ratio/Count Hancock clusters malware files into groups based on their byte-level histogram representation. It then counts the number of groups to which the files in X belong. If this number divided by the number of files in X exceeds a threshold ratio, or if the number exceeds a threshold count, Hancock rejects S. These files cannot be variants of a single malware family, if each malware group indeed corresponds to a malware family.

Signature Position Deviation Hancock calculates the position of S within each file in X, and computes the standard deviation of S's positions in these files. If the standard deviation exceeds a threshold, Hancock rejects S, because a large positional deviation suggests that S is included in the files it covers for very different reasons. Therefore, these files are unlikely to belong to the same malware family. The position of S in a malware file can be an absolute byte offset, which is with respect to the beginning of the file, or a relative byte offset, which is with respect to the beginning of the code section containing S.

Multiple Common Signatures Hancock attempts to find another common signature that is present in all the files in X and is at least 1 Kbyte away from S.

If such a common signature indeed exists and the distance between this signature and S has low standard deviation among the files in X, then Hancock accepts S because this suggests the files in X share a large chunk of code and thus are likely to be variants of a single malware family. Intuitively, this heuristic measures the similarity among files in X using additional signatures that are sufficiently far away, and can be generalized to using the third or fourth signature.

Surrounding Context Count Hancock expands S in each malware file in X by adding bytes to its beginning and end until the resulting byte sequences become different. For each such distinct byte sequence, Hancock repeats the same expansion procedure until the expanded byte sequences reach a size limit, or when the total number of distinct expanded byte sequences exceeds a threshold. If this expansion procedure terminates because the number of distinct expanded byte sequences exceeds a threshold, Hancock rejects S, because the fact that there are more than several distinct contexts surrounding S among the files in X suggests that these files do not belong to the same malware family.

5.2 Instruction-Level Diversity

Although byte-level diversity measurement techniques are easy to compute and quite effective in some cases, they treat bytes in a binary file as numerical values and do not consider their semantics. Given a signature S and the set of files it covers, X, instruction-level diversity measurement techniques, on the other hand, measure the instruction-level similarity or diversity among the files in X by extracting the instruction-level context surrounding S and computing the similarity among these contexts.

Enclosing Function Count Hancock extracts the enclosing function of S in each malware file in X, and counts the number of distinct enclosing functions. If the number of distinct enclosing functions of S with respect to X is higher than a threshold, Hancock rejects S, because S appears in too many distinct contexts among the files in X and therefore is not likely to be an intrinsic part of one or a very small number of malware families. To determine if two enclosing functions are distinct, Hancock uses the following three identicalness measures, in decreasing order of strictness:

- The byte sequences of the two enclosing functions are identical.
- The instruction op-code sequences of the two enclosing functions are identical. Hancock extracts the op-code part of every instruction in a function, and normalizes variants of the same op-code class into their canonical op-code. For example, there are about 10 different X86 op-codes for ADD, and Hancock translates all of them into the same op-code. Because each instruction's operands are ignored, this measure is resistant to intentional or accidental polymorphic transformations such as re-locationing, register assignment, etc.
- The instruction op-code sequences of the two enclosing functions are identical after *instruction sequence normalization.* Before comparing two op-code sequences, Hancock performs a set of de-obfuscating normalizations

that are designed to undo simple obfuscating transformations, such as replacing "`test esi, esi`" with "`or esi, esi`", replacing "`push ebp; mov ebp, esp`" with "`push ebp; push esp; pop ebp`", etc.

Enclosing Subgraph Count Hancock extracts a subgraph of the function call graph of every file in X centered at the call graph node corresponding to S's enclosing function, and compares the number of distinct enclosing subgraphs surrounding S. An enclosing subgraph is N-level if it contains up to N-hop neighbors of S's enclosing function in the function call graph. In practice, N is set to either 1 or 2. If the number of distinct N-level enclosing subgraphs of S with respect to X is higher than a threshold, Hancock rejects S, because S is not likely to be an intrinsic part of one or a very small number of malware families. This approach defines the surrounding context of S based on the set of functions that directly or indirectly call or are called by S's enclosing function, and their calling relationships. To abstract the body of the functions in the enclosing subgraphs, Hancock labels each node as follows: (1) If a node corresponds to a library function, Hancock uses the library function's name as its label. (2) If a node corresponds to a non-library function, Hancock labels it with the sequence of known API calls in the corresponding function. After labeling the nodes, Hancock considers two enclosing subgraphs as distinct if the edit distance between them is above a certain threshold.

Call Graph Cluster Count Hancock extracts the complete function call graph associated with every file in X, where each call graph node corresponds to either a non-library function or an entry-point library function, and partitions these graphs into different clusters according to their topological structure. Functions internal to a library are ignored. Nodes are labeled in the same way as described above. The assumption here is that the function call graphs of variants of a malware family are similar to one another, but the function call graphs of different malware families are distinct from one another. The distance threshold used in graph clustering is adaptively determined based on the average size of the input graphs. If the number of clusters obtained this way is higher than a threshold, Hancock rejects S because S seems to covers too many malware families and is thus likely to be an FP.

In summary, the byte-level counterparts of the enclosing function count, the enclosing subgraph count and the call graph cluster count are the surrounding context count, the deviation in signature position, and malware group count, respectively. The byte-level multiple common signature heuristic is a sampling technique to determine if the set of malware files covering a signature share a common context surrounding the signature.

6 Multi-Component String Signature Generation

Traditionally, string signatures used in AV scanners consist of a contiguous sequence of bytes. We refer to these as single-component signature (SCS). A natural generalization of SCS is multi-component signatures (MCS), which consist of multiple byte sequences that are potentially disjoint from one another. For example, we can use a 48-byte SCS to identify a malware program; for the same amount of storage space, we can create a two-component MCS with two 24-byte sequences. Obviously, an N-byte SCS is a special case of a K-component MCS where each component is of size $\frac{N}{K}$. Therefore, given a fixed storage space budget, MCS provides more flexibility in choosing malware-identifying signatures than SCS, and is thus expected to be more effective in improving coverage without increasing the false positive rate.

In the most general form, the components of a MCS do not need to be of the same size. However, to limit the search space, in the Hancock project we explore only those MCSs that have equal-sized components. So the next question is how many components a MCS should have, given a fixed space budget. Intuitively, each component should be sufficiently long so that it is unlikely to match a random byte sequence in binary programs by accident. On the other hand, the larger the number of components in a MCS, the more effective it is in eliminating false positives. Given the above considerations and the practical signature size constraint, Hancock chooses the number of components in each MCS to be between 3 and 5.

Hancock generates the candidate component set using a goodware model and a goodware set. Unlike SCS, candidate components are drawn from both data and code, because intuitively, combinations of code component signatures and data component signatures make perfectly good MCS signatures. When Hancock examines an $\frac{N}{K}$-byte sequence, it finds the longest substring containing this sequence that is common to all malware files that have the sequence. Hancock takes only one candidate component from this substring. It eliminates all sequences that occur in the goodware set and then takes the sequence with the lowest model probability. Unlike SCS, there is no model probability threshold.

Given a set of qualified component signature candidates, S1, and the set of malware files that each component signature candidate covers, Hancock uses the following algorithm to arrive at the final subset of component signature candidates used to form MCSs, S2:

1. Compute for each component signature candidate in S1 its *effective coverage value*, which is a sum of weights associated with each file the component signature candidate covers. The weight of a covered file is equal to its *coverage count*, the number of candidates in S2 already covering it, except when the number of component signatures in S2 covering that file is larger than or equal to K, in which case the weight is set to zero.
2. Move the component signature candidate with the highest effective coverage value from S1 to S2, and increment the coverage count of each file the component signature candidate covers.
3. If there are still malware files that are still uncovered or there exists at least one component signature in S1 whose effective coverage value is non-zero, go to Step 1; otherwise exit.

The above algorithm is a modified version of the standard greedy algorithm for the *set covering* problem. The only difference is that it gauges the value of each component signature candidate using its effective coverage value, which takes into account the fact that at least K component signatures in S2 must match a malware file before the file is considered covered. The way weights are assigned to partially covered files is meant to reflect the intuition that the value of a component signature candidate to a malware file is higher when it brings the file's coverage count from $X-1$ to X than that from $X-2$ to $X-1$, where X is less than or equal to K.

After S2 is determined, Hancock finalizes the K-component MCS for each malware file considered covered, i.e., whose coverage count is no smaller than K. To do so, Hancock first checks each component signature in S2 against a goodware database, and marks it as an FP if it matches some goodware file in the database. Then Hancock considers all possible K-component MCSs for each malware file and chooses the one with the smallest number of components that are an FP. If the number of FP components in the chosen MCS is higher than a threshold, T_{FP}, the MCS is deemed as unusable and the malware file is considered not covered. Empirically, T is chosen to be 1 or 2. After each malware file's MCS is determined, Hancock applies the same diversity principle to each MCS based on the malware files it covers.

There are several possible enhancements to the candidate signature filtering algorithm described above. First, one can put an inter-component distance constraint on the component signatures of a MCS, e.g., the byte distance between two consecutive component signatures of a MCS must be greater than a certain value in each malware file the MCS covers. As the threshold is increased, the files covered by a MCS are more likely to be similar to one another, and the MCS is less likely to be an FP. Second, one can put a content constraint on a MCS's component signatures as a whole, e.g., the number of data component signatures in each MCS must be smaller than $K-1$. This ensures that a MCS contains at least a certain number of code component signatures that are not an FP.

The current Hancock system uses a variant of the above algorithm. At each step, it starts with the component signature that covers the most uncovered files, then finds the component signature whose file coverage overlaps with the first component signature and that covers the most uncovered files, then the component signature whose file coverage overlaps with the first and second component signatures and that covers the most uncovered files, etc., until it finds K component signatures. Then it moves these K component signatures from S1 to S2 at a time, if these K component signatures collectively cover at least one uncovered file.

7 Evaluation

7.1 Methodology

To evaluate the overall effectiveness of Hancock, we used it to generate 48-byte string signatures for two sets of malware files, and use the coverage and number of false positives of these signatures as the performance metrics. The first malware set has 2,363 unpacked files that Symantec gathered in August 2008. The other has 46,288 unpacked files (or 112,156 files before unpacking) gathered in 2007-2008. The goodware model used in initial signature candidate filtering is derived from a 31-Gbyte goodware training set. In addition, we used another 1.8-Gbyte goodware set to filter out FP-prone signature candidates. To determine which signatures are FPs, we tested each generated signature against a 213-Gbyte goodware set. The machine used to perform these experiments has four quad-core 1.98-GHz AMD Opteron processors and 128 Gbytes of RAM.

7.2 Single-Component Signatures

Because almost every signature candidate selection and filtering technique in Hancock comes with an empirical threshold parameter, it is impossible to present results corresponding to all possible combinations of these parameters. Instead, we present results corresponding to three representative settings, which are shown in Table 1 and called *Loose*, *Normal* and *Strict*. The generated signatures cover overlapping sets of malware files.

To gain additional assurance that Hancock's FP rate was low enough, Symantec's malware analysts wanted to see not only zero false positives, but also that the signatures look good – they look like they encode non-generic behavior that is unlikely to show up in goodware. To that end, we manually ranked signatures on the August 2008 malware set as good, poor, and bad. Appendix A shows examples of good and bad signatures.

To get a rough indication of the maximum possible coverage, the last lines in tables 2 and 3 show the coverage of all non-FP candidate signatures. The probability-based and dissassembly-based heuristics were still enabled with Loose threshold settings.

Threshold setting	**Model probability**	**Group ratio**	**Position deviation**	**# common signatures**	**Interestingness**	**Minimum coverage**
Loose	-90	0.35	4000	1	13	3
Normal	-90	0.35	3000	1	14	4
Strict	-90	0.35	3000	2	17	4

Table 1. Heuristic threshold settings

Threshold setting	**Cover-age**	**# FPs**	**Good sig.s**	**Poor sig.s**	**Bad sig.s**
Loose	15.7%	0	6	7	1
Normal	14.0%	0	6	2	0
Strict	11.7%	0	6	0	0
All non-FP	22.6%	0	10	11	9

Table 2. Results for August 2008 data

Threshold	**Coverage**	**Sig.s**	**FPs**
Loose	14.1%	1650	7
Normal	11.7%	767	2
Normal, pos. deviation 1000	11.3%	715	0
Strict	4.4%	206	0
All non-FP	31.7%	7305	0

Table 3. Results for 2007-8 data

These results show not only that Hancock has a low FP rate, but also that tighter thresholds can produce signatures that look less generic. Unfortunately, it can only produce signatures to cover a small fraction of the specified malware.

Several factors limit Hancock's coverage:

- Hancock's packer detection might be insufficient. PEiD recognizes many packers, but by no means all of them. Entropy detection can also be fooled: some packers do not compress the original file's data, but only obfuscate it. Diversity-based heuristics will probably reject most candidate signatures extracted from packed files. (Automatically generating signatures for packed files would be bad, anyway, since they would be signatures on packer code.)
- Hancock works best when the malware set has many malware families and many files in each malware family. It needs many families so that diversity-based heuristics can identify generic or rare library code that shows up in several malware families. It needs many files in each family so that diversity-based heuristics can identify which candidate signatures really are characteristic of a malware family. If the malware sets have many malware families with only a few files each, this would lower Hancock's coverage.
- Malware polymorphism hampers Hancock's effectiveness. If only some code is polymorphic, Hancock can still identify high coverage signatures in the remaining code. If the polymorphic code has a relatively small number of variations, Hancock can still identify several signatures with moderate coverage that cover most files in the malware family. If all code is polymorphic, with a high degree of variation, Hancock will cover very few of the files.
- Finally, the extremely stringent FP requirement means setting heuristics to very conservative thresholds. Although the heuristics have good discrimination power, they still eliminate many good signatures. e.g. The group count heuristic clusters malware into families based on a single-byte histogram. This splits most malware families into several groups, with large malware families producing a large number of groups. An ideal signature for this family will occur in all of those groups. Thus, for the sake of overall discrimination power, the group count heuristic will reject all such ideal signatures.

Sensitivity Study A heuristic's *discrimination power* is a measure of its effectiveness. A heuristic has good discrimination power if the fraction of false positive signatures that it eliminates is higher than the fraction of true positive signatures it eliminates. These results depend strongly on which other heuristics are in use. We tested heuristics in two scenarios: we measured their *raw discrimination power* with other heuristics disabled; and we measured their *marginal discrimination power* with other heuristics enabled with conservative thresholds.

First, using the August 2008 malware set, we tested the raw discrimination power of each heuristic. Table 4 shows the baseline setting, more conservative setting, and discrimination power for each heuristic. The library heuristics (Universal FLIRT, library function reference, and address space) are enabled for the baseline test and disabled to test their own discrimination powers. Using all baseline settings, the run covered 551 malware files with 220 signatures and 84 false positives. Discrimination power is calculated as $\log \frac{\text{FPs}_i}{\text{FPs}_f} \Big/ \log \frac{\text{Coverage}_i}{\text{Coverage}_f}$.

Table 4 shows most of these heuristics to be quite effective. Position deviation and group ratio have excellent discrimination power (DP); the former lowers coverage very little and the latter eliminates almost all false positives. Model probability and code interestingness showed lower DP because their baseline settings were already somewhat conservative. Had we disabled these heuristics entirely, the baseline results would have been so overwhelmed with false positives as to be meaningless. All four of these heuristics are very effective.

Increasing the minimum number of malware files a signature must cover eliminates many marginal signatures. The main reason is that, for lower coverage numbers, there are so many more candidate signatures that some bad ones will get through. Raising the minimum coverage can have a bigger impact in combination with diversity-based heuristics, because those heuristics work better with more files to analyze.

Requiring two common signatures eliminated more good signatures than false positive signatures. It actually made the signatures, on average, worse.

Finally, the library heuristics all work fairly well. They each eliminate 50% to 70% of false positives while reducing coverage less than 30%. In the test for each library heuristic, the other two library heuristics and basic FLIRT functionality were still enabled. This shows that none of these library heuristics are redundant and that these heuristics go significantly beyond what FLIRT can do.

Heuristic	**FPs**	**Cov.**	**DP**
Max pos. deviation (from ∞ to 8,000)	41.7%	96.6%	25
Min file coverage (from 3 to 4)	6.0%	83.3%	15
Group ratio (from 1.0 to .6)	2.4%	74.0%	12
Model log probability (from -80 to -100)	51.2%	73.7%	2.2
Code interestingness (from 13 to 15)	58.3%	78.2%	2.2
Multiple common sig.s (from 1 to 2)	91.7%	70.2%	0.2
Universal FLIRT	33.1%	71.7%	3.3
Library function reference	46.4%	75.7%	2.8
Address space	30.4%	70.8%	3.5

Table 4. Raw Discrimination Power

Heuristic	**FPs**	**Coverage**
Max pos. deviation (from 3,000 to ∞)	10	121%
Min file coverage (from 4 to 3)	2	126%
Group ratio (from 0.35 to 1)	16	162%
Model log probability (from -90 to -80)	1	123%
Code interestingness (from 17 to 13)	2	226%
Multiple common sig.s (from 2 to 1)	0	189%
Universal FLIRT	3	106%
Library function reference	4	108%
Address space	3	109%

Table 5. Marginal Discrimination Power

Marginal Contribution of Each Technique Then we tested the effectiveness of each heuristic when other heuristics were set to the Strict thresholds from table 1. We tested the tunable heuristics with the 2007-8 malware set with Strict baseline threshold settings from table 1. Testing library heuristics was more computationally intensive (requiring that we reprocess the malware set), so we tested them on August 2008 data with baseline Loose threshold settings. Since both sets of baseline settings yield zero FPs, we decreased each heuristic's threshold (or disabled it) to see how many FPs its conservative setting eliminated and how much it reduced malware coverage. Table 5 shows the baseline and more liberal settings for each heuristic. Using all baseline settings, the run covered 1194 malware files with 206 signatures and 0 false positives.

Table 5 shows that almost all of these heuristics are necessary to reduce the FP rate to zero. Among the tunable heuristics, position deviation performs the best, eliminating the second most FPs with the lowest impact on coverage. The group ratio also performs well. Requiring a second common signature does not seem to help at all. The library heuristics perform very well, barely impacting coverage at all. Other heuristics show significantly decreased marginal discrimination power, which captures an important point: if two heuristics eliminate the same FPs, they will show good raw discrimination power, but poor marginal discrimination power.

7.3 Single-Component Signature Generation Time

The most time-consuming step in Hancock's string signature generation process is goodware model generation, which, for the model used in the above experiments, took approximately one week and used up all 128 GBytes of available memory in the process of its creation. Fortunately, this step only needs to be done once. Because the resulting model is much smaller than the available memory in the testbed machine, using the model to estimate a signature candidate's occurrence probability does not require any disk I/O.

The three high-level steps in Hancock at run time are malware pre-processing (including unpacking and disassembly), picking candidate signatures, and applying diversity-based heuristics to arrive at the best ones. Among them, malware pre-processing is the most expensive step, but is also quite amenable to parallelization. The two main operations in malware pre-processing are recursively unpacking malware files and disassembling both packed and unpacked files using IDA Pro. Both use little memory, so we parallelized them to use 15 of our machines 16 cores. For the 2007-2008 data set, because of the huge number of packed malware files and the decreasing marginal return of analyzing them, Hancock disassembled only 5,506 packed files. Pre-processing took 71 hours.

Picking candidate signatures took 145 minutes and 37.4 GB of RAM. 15 minutes and 34.3 GB of RAM went to loading the goodware model. The remainder was for scanning malware files and picking and storing candidate signatures in memory and then on disk.

Generating the final signature set took 420 minutes and 6.07 GB of RAM. Most of this time was spent running IDA Pro against byte sequences surrounding the final signatures to output their assembly representation. Without this step, the final signature generation step would have taken only a few minutes.

7.4 Multi-Component Signatures

We tested MCS signatures with 2 to 6 components, with each part being 16 bytes long. We used a 3.0 GB goodware set to select component candidates and tested for false positives with a 34.9 GB set of separate goodware.[2] Table 6 shows the coverage and false positive rates when 0 or 1 components could be found in the smaller goodware set.

# components	Permitted component FPs	Coverage	# Signatures	# FPs
2	1	28.9%	76	7
2	0	23.3%	52	2
3	1	26.9%	62	1
3	0	24.2%	44	0
4	1	26.2%	54	0
4	0	18.1%	43	0
5	1	26.2%	54	0
5	0	17.9%	43	0
6	1	25.9%	51	0
6	0	17.6%	41	0

Table 6. Multi-Component Signature results

We first observe that permitting a single component of an MCS to be an FP in our small goodware set consistently results in higher coverage. However, from 2- and 3-component signatures, we also see that allowing a single component FP results in more entire MCS FPs, where all signature components occur in a single goodware file.

We can trade off coverage and FP rate by varying the number of signatures components and permitted component FPs. Three to five part signatures with 0 or 1 allowed FPs seems to provide the best tradeoff between coverage and FPs.

Since we applied so few heuristics to get these results, beyond requiring the existence of the multiple, disjoint signature components which make up the signature, it is perhaps surprising that we have so few MCS FPs. We explain this by observing that although we do not limit MCS components to code bytes, we do apply all the library code reducing heuristics through IDA disassembly described in Section 4.2.

Also, the way in which signature components are selected from contiguous runs of identical bytes may reduce the likelihood of FPs. If a long, identical byte sequence exists in a set of files, the 16 byte signature component with lowest probability will be selected. Moreover, no other signature component will be selected from the same run of identical bytes. Thus, if malware shares an identical uncommon library (which we fail to identify as a library) linked in contiguously in the executable, at most one signature component will be extracted from this sequence of identical bytes. The other components must come from some other shared code or data.

Finding candidate signatures took 1,278 minutes and 117 GB of RAM. Picking the final signature sets took 5 to 17 minutes and used 9.0 GB of RAM.

[2] This final goodware set was smaller than in SCS tests because of the difficulty of identifying shorter, 16-byte sequences.

7.5 Comparison of Multi-Component Signatures with Single Component Signatures

Comparing our best coverage with no FPs for MCS to single-component signatures using our best combination of heuristics, we see that we get approximately double the coverage with MCS signatures. Moreover, unlike single signatures, we used few heuristics to get these results, beyond requiring the existence of the multiple signature components which make up the signature. Future work could include applying the heuristics developed for single component signatures to MCS, with the understanding that some heuristics (like the interestingness heuristic) will be difficult to apply on a sequence of only 16 bytes.

The MCS with more than 3 parts require more memory to store than the corresponding single component signatures. Depending on the scanning architecture, it may also be slower to scan for MCS signatures than single component signatures.

The final FP check uses a smaller goodware set than the one used for SCS because we had to build a more finely indexed data structure to support queries for the shorter, 16-byte sequences. Future work should include re-indexing the larger SCS goodware set.

More manual analysis of single component signatures was performed, and the heuristics were tightened beyond the point where the run had zero FPs, until the signatures looked good by manual analysis. A similar, in-depth analysis of MCS was not performed.

8 Discussion

The main limitation of the current version of Hancock is its low coverage, which is also the biggest surprise in this project. One potential explanation for this result is that malware authors have recently evolved their malware distribution strategy from a "few malware families each with many variants" model to a "many malware families each with few variants" model, so as to keep each distributed malware sample effective for as long as possible. Because Hancock is designed to generate string signatures that correspond to common byte sequences shared by variants of the same malware family, if the average number of variants in each family is decreased, it is more difficult for Hancock to generate signature with good coverage while keeping the false positive rate in check, especially when state-of-the-art malware classification technology is still quite primitive.

To generate new malware families, malware authors use sophisticated packing and/or metamorphic transformation tools. The current version of Hancock cannot do much for binaries created by these tools. The static unpack engine Hancock uses is used in Symantec's anti-virus products. Still it cannot handle many packers or metamorphic transformation tools. For example, in the largest test described in Section 6.2, Hancock has to ignore 59% of the input malware set because it found them to be packed and could not unpack them. Among the remaining 41%, some of them are probably packed (perhaps partially), but are not detected by Hancock. For such malware files, Hancock won't create string signatures for them because they do not share common byte sequences with other malware files.

In the future, we plan to incorporate dynamic unpacking techniques, such as Justin [25], to reduce the impact of packers on Hancock's coverage. It is also possible to mitigate the packer problem by blacklisting binaries packed by certain packers. We did not spend much effort investigating metamorphic transformation tools in the Hancock project, because string signature-based malware identification may not be effective for metamorphic binaries. Instead, behavior-based malware identification may be a more promising solution. Nonetheless, systematically studying modern metamorphic tools and devising a taxonomical framework to describe them will be very useful contributions to the field of malware analysis.

Another significant limitation of Hancock is its lack of dynamic analysis, which forces it to give up on packed or metamorphically transformed binaries that it cannot recognize or restore. The rationale for the design decision of employing only static analysis in Hancock is that it cannot afford the run-time performance cost associated with dynamic analysis given the current and future malware arrival rate. In addition, even state-of-the-art dynamic analysis techniques cannot solve all the packer or metamorphism problems for Hancock.

Although many of Hancock's heuristics can be evaded, in general this is a much smaller concern than the problem that malware authors avoid using known string signatures in their binaries. Attackers can (and do) test newly generated malware files against popular anti-virus products. In contrast, even if malware authors create malware files that do not contain byte sequences that Hancock may use as signatures, there is no easy way to test the effectiveness of these malware files against Hancock's signature generation algorithms, because it is not publicly available and because it has so many empirical built-in parameters. In theory, security by obscurity is not a foolproof solution; in practice, it is very difficult, if not infeasible, to evade Hancock's signature generation heuristics.

9 Conclusion

Given a set of malware files, an ideal string signature generation system should be able to automatically generate signatures in such a way that the number of signatures required to cover the malware set is minimal and the probability of these signatures appearing in goodware programs is also minimal. The main technical challenge of building such string signature generation systems is how to determine how FP-prone a byte sequence is without having access to even a sizeable portion of the world's goodware set. This false positive problem is particularly challenging because the goodware set is constantly growing, and is potentially unbounded. In the Hancock project, we have developed a series of signature selection and filtering techniques that collectively could remove most, if not all, FP-prone signature candidates, while maintaining a reasonable coverage of the input malware set. In summary, the Hancock project has made the following research contributions in the area of malware signature generation:

- A scalable goodware modeling technique that prunes away unimportant nodes according to their relative information gain and merges sub-models so as to scale to very large training goodware sets,
- A set of diversity-based techniques that eliminate signature candidates when the set of malware programs they cover exhibit high diversity, and
- The first known string signature generation system that is capable of creating multi-component string signatures which have been shown to be more effective than single-component string signatures.

Although Hancock represents the state of the art in string signature generation technology, there is still room for further improvement. The overall coverage of Hancock is lower than what we expected when we started the project. How to improve Hancock's coverage without increasing the FP rate of its signatures is worth further research. Although the multi-component signatures that Hancock generates are more effective than single-component signatures, their actual run-time performance impact is unclear and requires more thorough investigation. Moreover, there could be other forms of multi-component signatures that Hancock does not explore and therefore deserve additional research efforts.

References

1. PEiD. http://www.peid.info
2. Clam AntiVirus: Creating signatures for ClamAV. http://www.clamav.net/doc/latest/signatures.pdf (2007)
3. Arnold, W., Tesauro, G.: Automatically generated win32 heuristic virus detection. In: Proceedings of VIRUS BULLETIN CONFERENCE. (2000)
4. Jacob, G., Debar, H., Filiol, E.: Behavioral detection of malware: from a survey towards an established taxonomy. Journal in Computer Virology **4**(3) (2008)
5. Singh, S., Estan, C., Varghese, G., Savage, S.: Automated worm fingerprinting. In: OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation, Berkeley, CA, USA, USENIX Association (2004) 4–4
6. Kim, H.: Autograph: Toward automated, distributed worm signature detection. In: Proceedings of the 13th Usenix Security Symposium. (2004) 271–286
7. Kreibich, C., Crowcroft, J.: Honeycomb: creating intrusion detection signatures using honeypots. SIGCOMM Comput. Commun. Rev. **34**(1) (January 2004) 51–56
8. Newsome, J., Karp, B., Song, D.: Polygraph: Automatically generating signatures for polymorphic worms. In: SP '05: Proceedings of the 2005 IEEE Symposium on Security and Privacy, Washington, DC, USA, IEEE Computer Society (2005) 226–241
9. Li, Z., Sanghi, M., Chen, Y., Kao, M., Chavez, B.: Hamsa: fast signature generation for zero-day polymorphic worms with provable attack resilience. In: SP 06: Proceedings of the 2006 IEEE Symposium on Security and Privacy (Oakland06, IEEE Computer Society (2006) 32–47

10. Tang, Y., Chen, S.: Defending against internet worms: A signature-based approach. In: Proceedings of IEEE INFOCOM05. (2005)
11. Christodorescu, M., Jha, S., Seshia, S., Song, D., Bryant, R.: Semantics-aware malware detection. In: Proceedings of the IEEE Symposium on Security and Privacy. (2005)
12. Yegneswaran, V., Giffin, J.T., Barford, P., Jha, S.: An architecture for generating semantics-aware signatures. In: SSYM'05: Proceedings of the 14th conference on USENIX Security Symposium, Berkeley, CA, USA, USENIX Association (2005) 7–7
13. Newsome, J., Song, D.: Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In: Proceedings of the Network and Distributed System Security Symposium (NDSS 2005). (2005)
14. Costa, M., Crowcroft, J., Castro, M., Rowstron, A., Zhou, L., Zhang, L., Barham, P.: Vigilante: end-to-end containment of internet worms. In: SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles, New York, NY, USA, ACM (2005) 133–147
15. Xu, J., Ning, P., Kil, C., Zhai, Y., Bookholt, C.: Automatic diagnosis and response to memory corruption vulnerabilities. In: CCS '05: Proceedings of the 12th ACM conference on Computer and communications security, New York, NY, USA, ACM (2005) 223–234
16. Liang, Z., Sekar, R.: Automatic generation of buffer overflow attack signatures: An approach based on program behavior models. In: ACSAC '05: Proceedings of the 21st Annual Computer Security Applications Conference, Washington, DC, USA, IEEE Computer Society (2005) 215–224
17. Liang, Z., Sekar, R.: Fast and automated generation of attack signatures: a basis for building self-protecting servers. In: CCS '05: Proceedings of the 12th ACM conference on Computer and communications security, New York, NY, USA, ACM (2005) 213–222
18. Wang, X., Li, Z., Xu, J., Reiter, M.K., Kil, C., Choi, J.Y.: Packet vaccine: black-box exploit detection and signature generation. In: CCS '06: Proceedings of the 13th ACM conference on Computer and communications security, New York, NY, USA, ACM (2006) 37–46
19. Brumley, D., Newsome, J., Song, D., Wang, H., Jha, S.: Towards automatic generation of vulnerability-based signatures. In: SP '06: Proceedings of the 2006 IEEE Symposium on Security and Privacy, Washington, DC, USA, IEEE Computer Society (2006) 2–16
20. Wang, H.J., Guo, C., Simon, D.R., Zugenmaier, A.: Shield: Vulnerability-driven network filters for preventing known vulnerability exploits. In: ACM SIGCOMM. (2004) 193–204
21. Cui, W., Peinado, M., Wang, H.J., Locasto, M.E.: Shieldgen: Automatic data patch generation for unknown vulnerabilities with informed probing. In: SP '07: Proceedings of the 2007 IEEE Symposium on Security and Privacy, Washington, DC, USA, IEEE Computer Society (2007) 252–266
22. Kephart, J.O., Arnold, W.C.: Automatic extraction of computer virus signatures. In: Proceedings of the 4th Virus Bulletin International Conference. (1994)
23. Begleiter, R., El-Yaniv, R., Yona, G.: On prediction using variable order markov models. Journal of Artificial Intelligence Research **22** (2004) 384–421
24. Guilfanov, I.: Fast library identification and recognition technology. http://www.hex-rays.com/idapro/flirt.htm (1997)
25. Guo, F., Ferrie, P., Chiueh, T.: A study of the packer problem and its solutions. Recent Advances in Intrusion Detection (2008) 98–115

A Example Signatures

A.1 Good Signature

Figure 3 shows the best signature generated from Section 7.2 for August 2008 malware. Hancock accepts this signature, even with the most conservative heuristic thresholds.

Two features make this an excellent signature. First, it uses 16-bit registers, which is quite rare in goodware. Second, it has 8 constants with different, unusual values. Intuitively, this looks like an excellent signature.

Its heuristic scores are also excellent: its probability is 2^{-140}, versus the Strict threshold of 2^{-90}; its interestingness score is 33, versus 17; and its diversity-based heuristic scores are perfect, with zero position deviation, a group count of one, and with second and third signatures identified by the multiple common signature heuristic. Finally, this signature covers 73 files, which is the highest of any signature in this set.

This is one signature where the automated heuristics match very well with manual assessment.

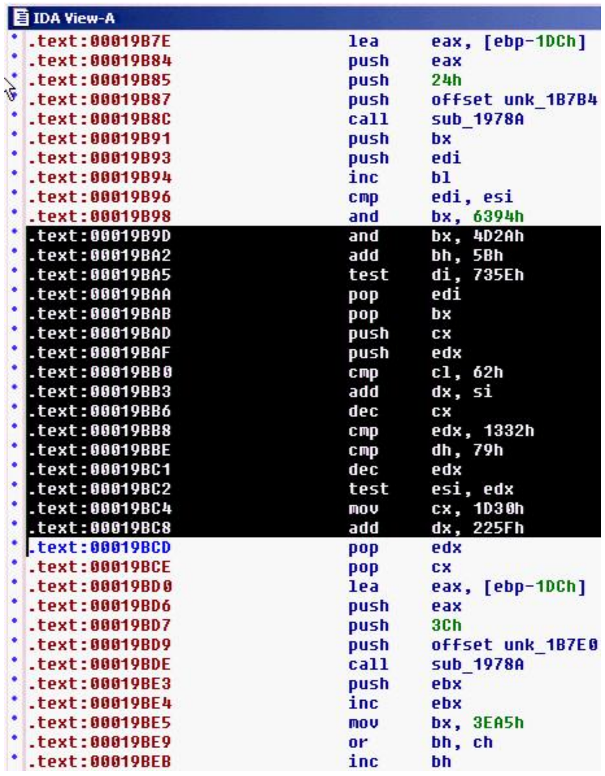
A.2 Bad Signature

Figure 4 shows a bad signature from the same set of generated signatures. Even the Loose thresholds (from table 1) reject it.

The logic for this signature looks generic. Its comparisons and jumps look like typical goodware. It has only one insteresting 1-byte constant. It follows a call to a common Win32 API. Intuitively, it looks pretty generic.

Many of the heuristic scores for this signature are marginal. Its probability is not bad, at 2^{-107}. Its interestingness score is 13, the lowest allowed by Loose thresholds. Its position deviation, 1851, is okay. Its group ratio, $\frac{6}{7}$, disqualifies it. The multiple common signature heuristic did not find any additional signatures. Finally, it covers only 7 files.

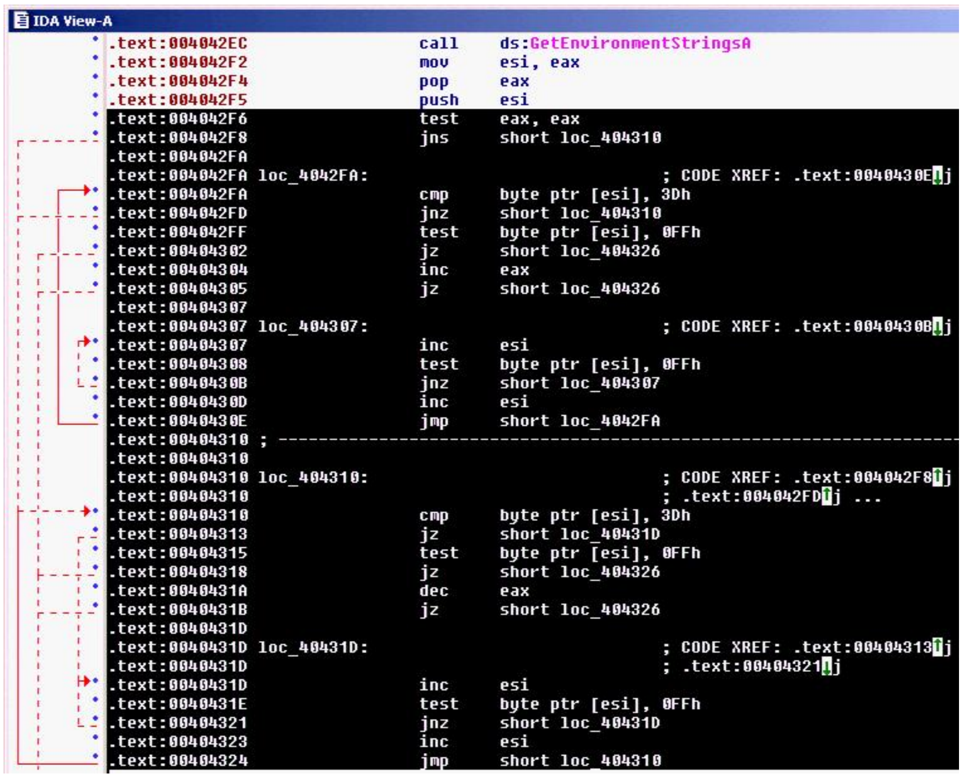
This signature is another case where automated heuristics match with manual assessment.



IDA View-A

```
.text:00019B7E                 lea     eax, [ebp-1DCh]
.text:00019B84                 push    eax
.text:00019B85                 push    24h
.text:00019B87                 push    offset unk_1B7B4
.text:00019B8C                 call    sub_1978A
.text:00019B91                 push    bx
.text:00019B93                 push    edi
.text:00019B94                 inc     bl
.text:00019B96                 cmp     edi, esi
.text:00019B98                 and     bx, 6394h
.text:00019B9D                 and     bx, 4D2Ah
.text:00019BA2                 add     bh, 5Bh
.text:00019BA5                 test    di, 735Eh
.text:00019BAA                 pop     edi
.text:00019BAB                 pop     bx
.text:00019BAD                 push    cx
.text:00019BAF                 push    edx
.text:00019BB0                 cmp     cl, 62h
.text:00019BB3                 add     dx, si
.text:00019BB6                 dec     cx
.text:00019BB8                 cmp     edx, 1332h
.text:00019BBE                 cmp     dh, 79h
.text:00019BC1                 dec     edx
.text:00019BC2                 test    esi, edx
.text:00019BC4                 mov     cx, 1D30h
.text:00019BC8                 add     dx, 225Fh
.text:00019BCD                 pop     edx
.text:00019BCE                 pop     cx
.text:00019BD0                 lea     eax, [ebp-1DCh]
.text:00019BD6                 push    eax
.text:00019BD7                 push    3Ch
.text:00019BD9                 push    offset unk_1B7E0
.text:00019BDE                 call    sub_1978A
.text:00019BE3                 push    ebx
.text:00019BE4                 inc     ebx
.text:00019BE5                 mov     bx, 3EA5h
.text:00019BE9                 or      bh, ch
.text:00019BEB                 inc     bh
```

Fig. 3. A good signature produced by the run from Section 7.2 on August 2008 malware


IDA View-A

```
.text:004042EC                 call    ds:GetEnvironmentStringsA
.text:004042F2                 mov     esi, eax
.text:004042F4                 pop     eax
.text:004042F5                 push    esi
.text:004042F6                 test    eax, eax
.text:004042F8                 jns     short loc_404310
.text:004042FA
.text:004042FA loc_4042FA:                             ; CODE XREF: .text:0040430E↓j
.text:004042FA                 cmp     byte ptr [esi], 3Dh
.text:004042FD                 jnz     short loc_404310
.text:004042FF                 test    byte ptr [esi], 0FFh
.text:00404302                 jz      short loc_404326
.text:00404304                 inc     eax
.text:00404305                 jz      short loc_404326
.text:00404307
.text:00404307 loc_404307:                             ; CODE XREF: .text:0040430B↓j
.text:00404307                 inc     esi
.text:00404308                 test    byte ptr [esi], 0FFh
.text:0040430B                 jnz     short loc_404307
.text:0040430D                 inc     esi
.text:0040430E                 jmp     short loc_4042FA
.text:00404310 ; ---------------------------------------------------------------
.text:00404310
.text:00404310 loc_404310:                             ; CODE XREF: .text:004042F8↑j
.text:00404310                                         ; .text:004042FD↑j ...
.text:00404310                 cmp     byte ptr [esi], 3Dh
.text:00404313                 jz      short loc_40431D
.text:00404315                 test    byte ptr [esi], 0FFh
.text:00404318                 jz      short loc_404326
.text:0040431A                 dec     eax
.text:0040431B                 jz      short loc_404326
.text:0040431D
.text:0040431D loc_40431D:                             ; CODE XREF: .text:00404313↑j
.text:0040431D                                         ; .text:00404321↓j
.text:0040431D                 inc     esi
.text:0040431E                 test    byte ptr [esi], 0FFh
.text:00404321                 jnz     short loc_40431D
.text:00404323                 inc     esi
.text:00404324                 jmp     short loc_404310
```

Fig. 4. A bad signature produced by the run from Section 7.2 on August 2008 malware