Heuristic Malware Detection via Basic Block Comparison

Francis Adkins  
Department of Computer Science  
United States Air Force Academy  
Francis.Adkins.1@us.af.mil

Luke Jones  
Department of Computer Science  
United States Air Force Academy  
Luke.Jones.2@us.af.mil

Abstract

Each day, malware analysts are tasked with more samples than they have the ability to analyze by hand. To produce this trend, malware authors often reuse a significant portion of their code. In this paper, we introduce a technique to statically decompose malicious software to identify shared code. This technique variably applies a sliding-window methodology to either full files or individual basic blocks to produce representative similarity ratios either between two binaries or between two functionalities within binaries, respectively. This grants the ability to apply heuristic detection via threshold similarity matching as well as full-inclusivity matching for malicious functionality. Additionally, we apply generalization techniques to minimize local assembly variants while still maintaining consistent structural matching. We also identify improvements that this technique provides over previous technologies and demonstrate its success in practical sample detection. Finally, we suggest further applications of this technique and highlight possible contributions to modern malware detection.

Distribution A. Approved for public release, distribution unlimited.

1. Introduction

1.1. History

In the infancy of personal computers, malware could be described as simple pranks among colleagues. However, what began as playful antics has progressed to a global multi-billion dollar industry that now poses a significant threat to corporate industry. In addition to potential leaks of customer information, businesses are also faced with intellectual property theft, competition-based intelligence gathering, and even the destruction of company resources. Annually, these cyber-attacks result in millions of dollars in damage [2, 3].

The year 2010 shocked the world again with the introduction of malware produced by groups known as Advanced Persistent Threats (APTs). The malware constructed by these entities have been renowned for their stealth capabilities, devastating payload, and highly targeted nature. The arrival of the Stuxnet worm put malware into the public spotlight as it damaged Iranian nuclear facilities and marked the first recorded physical damage caused by a state-sponsored malicious computer program. Shortly thereafter, related malware such as Duqu, Flame, and Gauss demonstrated just how prevalent these advanced attacks could be.

As malware has become more destructive, so too has it developed in stealth capability. The malware analyst’s job has become more difficult in recent years as packers, metamorphism, polymorphism, rootkits, and antivirus evasion techniques have made detection and analysis more difficult [7]. In a recent example, the Flame worm was able to hide itself by utilizing multiple rootkit techniques and actively search for more than a dozen antivirus products to evade detection [8]. Similarly, the rootkit capabilities of Stuxnet allowed it to remain undetected for years while its highly destructive payload exemplified the increasingly militaristic nature of malware [9]. These types of attacks herald the age of cyber-war and legitimize the offensive capabilities provided by such programs. These behaviors have become increasingly typical in the modern age of malware and necessitate new defense techniques. Yet, despite the extreme
potential for damage, no existing technology has demonstrated the capacity to meet these threats.

1.2. The Current Climate

The antivirus industry has attempted to address this challenge but has yet to have any major success. While antivirus products work well against known threats, they rarely have the ability to identify new specimens before they can act. Instead, this identification is left to human experts after the damage has been done - experts who are overwhelmed by the ever-growing torrent of new samples reported each day. Ultimately, this human aspect forms a nearly insurmountable bottleneck for the security industry as countless man-hours are spent fingerprinting new threats [4].

However, there are methods to combat these issues. While the number of malware samples is steadily growing, many new specimens are simply variants of existing samples. Additionally, a majority of malware samples employ similar techniques to ensure persistence and communication channels [5]. By highlighting similarities between malware and analyzing how this behavior differs from traditional, non-malicious programs, it is possible to categorize malicious software based simply on statically-contained functionality.

1.3. Contributions

In this paper, we make the following contributions:

- We propose an algorithm to identify shared executable code between binaries. This process involves generating a fingerprint for sections of code through a process of feature hashing. Fingerprint comparison then allows us to produce a reliable similarity ratio for further investigation.
- We highlight the improvements that our algorithm demonstrates over previous technologies with similar capabilities.
- We demonstrate our algorithm’s ability to detect malicious software in a variety of practical situations. These include: full-file matching, single-function identification, and malicious functionality inclusion.
- We propose future applications for our technique and suggest novel methods of malware detection and attribution.

2. Related Work

2.1. Bitshred

The Bitshred project has undergone several iterations in recent years. However, the original implementation features a sophisticated assembly-level code reuse detection algorithm that is extremely useful for determining if a given program is similar to other programs known to be malware [5]. The algorithm works by first “shredding” the full executable section of a binary - dividing it into n-byte length segments. All sheds of a file are then hashed into a Bloom filter which is used for memory-efficient set membership tests. Once the sheds have been added to the Bloom filter, the filter is considered the fingerprint of the file. Finally, Bitshred computes a Jaccard Index to quantify the level of similarity between Bloom filter fingerprints. A Jaccard Index simply returns the percent of similarity between two items. Since the Bloom filter fingerprints are effectively binary strings, a Jaccard Index is computed as so:

\[
J(A, B) = \frac{S(BF_A \cap BF_B)}{S(BF_A) + S(BF_B) - S(BF_A \cap BF_B)}
\] (1)

Where \(S(BF)\) indicates the number of set bits in the Bloom filter. However, this implementation of a Jaccard Index does not account for the case in which \(S(BFB)\) is smaller than \(S(BFA)\) and is contained within it. In this situation we must implement the slightly modified containment case:

\[
J(A, B) = \frac{S(BF_A \cap BF_B)}{S(BF_B)}
\] (2)

Interestingly, BitShred separated a sample of 9,404 known malware binaries into 2,895 unique clusters at the highest discrimination setting. This technique bears similarities to signature-based detection, due to the shared agnosticism towards the actual functionality of the binary. However, BitShred is undoubtedly both much more sophisticated than any signature-based technique and much more likely to produce useful information to detect malicious software [5].

2.2. Saebjørnsen et al.

Another recent algorithm produced by Saebjørnsen et al. is nameless but utilizes an approach that actually leverages the functionality of the binary to detect code clones [11]. First, the algorithm disassembles the binary. Then the algorithm normalizes the resulting assembly instructions by abstracting the operands of the instruction to generically remembered variables
typified by either register reference, memory location, or constant value. Next, the authors split clone detection into two parts: exact and inexact. For the exact portion, the algorithm only returns portions of code that have exactly the same normalized assembly instructions sequences. However, for inexact clone matching, feature vectors are calculated for sequences of assembly instructions and the algorithm groups similar feature vectors to return code clones. Next, the algorithm removes extraneous clones, such as clones which overlap on significant portions of code. Lastly, small adjacent clones are grouped into larger clones. The authors chose to test their algorithm on the system files of a Windows XP machine. Unsurprisingly, many Windows system dynamic libraries and executables have shared code, and the clones were to be expected (e.g., tasklist.exe and taskkill.exe were found to be similar). Malware analysts could easily apply this algorithm to viruses, Trojans, and other malware to find similarities and then use that information to identify binaries as malicious or benign [11].

2.3. SMIT

Researchers at Symantec labored on an idea that focused less on finesse of the analysis of the binary and more on refining an algorithm to find similar graphs and, therefore, similar programs. Their program is called the Symantec Malware Indexing Tree (SMIT). In the analysis of binaries, SMIT takes for granted that the in-house Symantec unpacker will work to reverse the packing algorithm of malware, including recursive operation through multiple layers of packing. Once the unpacker has effectively revealed the malware’s original code, the algorithm uses IDA Pro to identify and categorize functions. From all identified functions, SMIT builds the function-call graph. Next, SMIT tries to find similarities to malware for a given program by using graph-edit distances obtained via graph matching and optimizations such as the neighbor-biased Hungarian algorithm. SMIT was ultimately very successful and had a max malware family detection rate of 91.3% [4].

2.4. Others

A number of other projects have also been dedicated to identifying shared or similar code and are available for further research [13, 14, 15, 16, 17].

3. BBCP

3.1. Overview

In this section, we propose our algorithm, BBCP - the Basic Block Comparison Platform. The concept for this algorithm was derived from the projects mentioned in sections 2.1-2.3, above. It is an amalgamation of the original Bitshred project’s methodology, the generalization techniques described by Saebjørnsen et al., and the use of IDA Pro as described in the SMIT project.

BBCP operates in the following stages:

1. Dissect either the full executable section or basic blocks of a binary file into n-length “shreds” or n-grams.
2. Generate a fingerprint for the subject utilizing these shreds through a process of feature hashing.
3. Compare two shreds to determine shared feature inclusion and generate a similarity index.

The variability of this algorithm allows us to identify shared executable code in a variety of situations. First, full-executable-section comparison allows us to identify simple malware variants. This process will detect if two programs share mostly similar code without relying on direct signature comparison. Second, BBCP can also detect if malicious functionality is contained within an executable. This process can be used to detect ‘bad’ code within a predominantly ‘good’ executable, as could be the case in trojan horse malware. Finally, individual basic-block comparison allows us to flag software based simply on its inclusion of a known-malicious block.

3.2. Algorithm Functionality

BBCP is founded on the common reverse engineering tool IDA Pro as well as the scripting language it offers, IDAPython. The use of IDA allows us to target a number of platforms regardless of the operating system from which BBCP is run. It also contributes to the two modes of operation offered by BBCP: full-file dissection and basic block dissection.

3.3. Modes of Operation

In full-file mode, also known as shallow-dive dissection (SDD), BBCP analyzes the entire executable portion of a binary. This is identified via the section name (i.e. “code” or “text”) presented within IDA. Full-file mode is capable of comparing similar-size binaries and detecting slight variations among them.
This is representative of the case in which only slight changes were made to a malware sample to create a variant. Additionally, full-file mode is also able to identify containment scenarios, such as the case in which a malicious executable is located within overall non-malicious software (e.g. a trojan horse).

In basic block mode, also known as deep-dive dissection (DDD), BBCP instead only analyzes each individual basic block of the binary. For our purposes, this was represented by function boundaries within IDA. This process has a number of uses, including detection of known-malicious blocks as well as filtration of known-good blocks to identify unknown code. We will address these topics in greater depth later.

3.4. Shredding

The first step in the BBCP process is that of “shredding” [5]. This terminology is derived from Bitshred, from which BBCP was inspired. In this step, the appropriate section is broken into n-length shards via a sliding-window methodology. However, in the Bitshred project, these windows were derived from hexadecimal bytes across the full executable section of a binary. In contrast, BBCP instead derives windows from the full text instructions as obtained from IDA. This technique allows us to apply generalization to our results to mitigate the effects of code reordering, compiler optimizations, and, to a small extent, different compilers.

The normalization presented in BBCP operates through the following five transformations:

- Registers replaced with “REG”
- Locations replaced with “LOC”
- Constant memory references replaced with “MEM”
- Constant values replaced with “CONST”
- Variable references replaced with “VAR”

As suggested by [11], these normalization processes minimize the effects produced by some code variations. For example, in our implementation, these transformations minimize the influence of both variable and function reordering. Thus, our algorithm is somewhat more capable of detecting modern polymorphic and metamorphic malware, both of which heavily utilize these techniques. The full shredding and normalization process is represented in the following figure:

3.5. Feature Hashing

Large scale feature-to-feature comparison is heavily time consumptive and thus not practical in an operational implementation. Therefore, we instead add each shred to a feature hash to implement scalable set comparison. Each feature hash will, in turn, create a unique fingerprint for that particular section of executable code. We implement feature hashing as so:

a) Create a 0-initialized binary array (feature hash) of length $2^x$, where x is an accuracy factor explained below
b) Create an MD5 sum of the current shred
c) Retrieve the final x bits of the MD5 sum -- this is used as the feature hash index
d) Set $fh[x]$ = 1
e) Repeat steps b-d for all shreds

Because only a subset of the full hashing algorithm is used as our index, this method is subject to false-positives. However, in the course of experimentation, we have concluded that a feature hash size of $x = 2^{22}$ yields the most reasonable trade-off of size-on-disk versus collision false-positives.

3.6. Comparison

As a final step, BBCP computes a Jaccard Index to quantify the level of similarity between feature hashes. A Jaccard Index simply returns the percent of similarity between two items. Similar to the Bloom Filters used in the original Bitshred project, feature hashes are represented as binary strings. Thus, a
Jaccard Index is computed in the same manner described in [1].

In BBCP, the ultimate result of these comparisons is a representative similarity metric of shared code structure. This information can then be used in the process of malware detection.

3.7. Inapplicability of Usual Bloom Filters

Based on the original BitShred paper, we first re-implemented BitShred using Bloom filters instead of feature hashes as described above. A Bloom filter is intended to serve as a platform for memory-efficient set membership tests and is typically composed of multiple feature hashes combined into one data structure. That is to say, Bloom filters hash input values with multiple hashing algorithms and set all corresponding indices to “1”. When represented as a binary string, this raises the probability of bit-wise collisions extraordinarily high. Since our intent with this project is to execute bit-wise comparisons and not data structure dependent set-membership testing, the concept of feature hashes is more applicable than Bloom filters. A deeper discussion of this is available in [12].

This is mentioned because, in subsequent sections, BBCP is not compared to the original BitShred, but rather to BitShred using feature hashing instead of Bloom filters. Therefore, because the following tests use BBCP with feature hashing and BitShred with feature hashing, the comparison is between which types of windows yield more accurate results. In other words: are sliding windows on full assembly instruction boundaries composed of assembly code text better than sliding windows on byte-boundaries composed of bytes?

4. Bitshred Improved

4.1. Same Source, Different Compilers

The premise of this test is that compiling the same source with different compilers should yield different binaries, but whichever algorithm reports higher similarity is more correct. This follows because it would be highly desirable if a similarity detection algorithm was agnostic of the compiler used on the source code. Neither BBCP nor BitShred was made to have this agnosticism, but whatever reported the closest value to 100% would be the better algorithm.

For this test, we considered that the content of the shared code is irrelevant and thus experimented with a cryptographic library written in C++. We concluded that, in this situation, shared code is shared code, regardless of intent. Furthermore, many malware functions are cryptographic in nature and, thus, such a library should serve as an adequate test specimen.

To make debugging more accessible, only one hashing function with all of it dependencies was compiled. What remained of the cryptographic library was compiled into a dynamic link library. To have some metric for our tests, we dumped the n-grams that BBCP created for comparison and the n-grams that BitShred created for comparison, and conducted Linux command line operations on them to determine their respective similarity without using a feature hash. Non-unique n-grams were removed, the n-grams were sorted and then a series of Linux “comm” commands were done to see how many were the same between the binary compiled with Microsoft Visual Studio and the binary compiled with MinGW.

The following data and all other BBCP data in this paper were collected using a feature hash accuracy of 22 bits:

<table>
<thead>
<tr>
<th>Table 1. Same source, Different compilers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bitsched</td>
</tr>
<tr>
<td>Jaccard Index</td>
</tr>
<tr>
<td>Terminal Command Analysis</td>
</tr>
</tbody>
</table>

BitShred indicated 0% similarity using terminal command analysis because no two windows were exactly the same. Its Jaccard index still indicates some amount of similarity because the false positive rate for Bloom filters is very high. The bit array produced by hashing windows must be very sparse for the signature to be useful. BitShred creates around 30,000 n-grams when shredding the test executable, whereas BBCP creates only around 10,000 n-grams. This means that by the end of the operation, the likelihood of feature hash collisions is three times more likely in the case of BitShred than in the case of BBCP. This explains why there are many more collisions with BitShred, inflating the value of the Jaccard index, while BBCP appears to have no collisions which inflate the Jaccard index. The disparity between BBCP’s Jaccard index and its terminal command analysis value, 0.25%, is considered within the realm of probable error.

From these results, BBCP wins the test, though by only 1%. In actuality, two binaries being 1% similar is negligible, however, it is significant that BitShred, via
a sliding window of hexadecimal values on byte boundaries, produced no n-grams which were the same, while BBCP, via a sliding window of mnemonics and opcodes on instruction boundaries, did find some n-grams that were the same. We consider this more preferable than the alternative. However, the improvements of BBCP over BitShred are much more readily visible in the next test.

4.2. Source Subset, Same Compilers

The premise of this test was that some given source and a subset of that given source will compile to extremely similar binaries. In fact, the compiled binary of the source subset should be approximately 100% contained by the compiled binary of the whole source. “Approximate” is a key term since we found that certain functions, exactly akin in the source, were compiled slightly differently between the subset and the whole. For this test, we used the entirety of the cryptographic hash library mentioned in the above section. For the subset of the source, we removed one class and its header files and recompiled the project using the same compiler, Microsoft Visual Studio.

<table>
<thead>
<tr>
<th>Jaccard Index</th>
<th>BitShred</th>
<th>BBCP, SDD</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.37848196</td>
<td>0.94929424</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Jaccard Index Containment Case</th>
<th>BitShred</th>
<th>BBCP, SDD</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.55411762</td>
<td>0.97705984</td>
<td></td>
</tr>
</tbody>
</table>

Considering the Jaccard containment case, BBCP is right on the mark. In fact, because the source subset is approximately 100% contained by the original source, BBCP is almost twice as accurate as the BitShred algorithm. This is attributable to BBCP augmenting the meaning in n-grams by shifting BitShred’s focus from pure hex – which inevitably degrades meaning by unnaturally breaking up assembly instructions – to text-based assembly instructions themselves. Focusing on assembly instructions for n-grams also enabled their normalization, a process which boosted similarity values the most significantly of any of BBCP’s improvements over BitShred.

4.3. Same Source Embedding into Dissimilar Programs

The purpose of this test was to demonstrate BBCP’s ability to detect specific shared code between two binaries and show that BBCP indicates dissimilarity when appropriate. For this test, we retrieved two vastly dissimilar programs from MSDN’s sample code repository. We then embedded the netcat for Windows source code, as obtained from the JonCron.org blog, into each of these executables. Comparison yielded the following results:

<table>
<thead>
<tr>
<th>Table 3. Source embedding</th>
</tr>
</thead>
<tbody>
<tr>
<td>SDD - Jaccard Index (Non-Containment)</td>
</tr>
<tr>
<td>Comparison of Program A to Program B - Original</td>
</tr>
<tr>
<td>Comparison of Program A to Program B - With Netcat Containment Case</td>
</tr>
</tbody>
</table>

It is once again worth noting that two identical pieces of source code will not necessarily compile to the same assembly instructions within different binaries. In particular, it is often the case that one instruction will be replaced with a different, yet highly similar alternative (such as xor eax,eax for mov eax, 0). Thus, while we would ideally like all shared functionality to be 100% identical between our two subjects, this was not the case. Instead, through experimentation, a 65% threshold was adequate to identify the contained netcat functions.

The results above demonstrate that full-file matching (SDD) is not always the most helpful in identifying shared functionality. However, basic block matching (DDD) was able to identify not only that code was shared, but in which specific blocks. From this point, we can turn these specific blocks over to a malware analyst for further investigation - thus significantly lowering the time required to find suspicious code.

5. Conclusions and Future Work

5.1. Provenance Determination

One of the greatest challenges faced in cyber conflict today is the concept of provenance. With the rise in nation-state sponsored malwares, uncovering the roots of the infection has become more important than ever. Now, for the first time in history, discovering the beginnings of a piece of software can have international ramifications.
In May of 2012, Kaspersky Lab announced the discovery of the most sophisticated piece of malware since Stuxnet - the Flame worm [18]. Malware analysts worked feverishly for more than two weeks before finally uncovering the link that they had been searching for: a 100% identical module shared between Stuxnet and Flame [19]. This similarity, when correlated to the time at which the samples were collected, provided irrefutable evidence that Stuxnet and Flame had been developed by the same authors and during the same time period - prior to Stuxnet’s discovery. This revelation eventually exposed an entire platform from which the two pieces of malware seem to have been co-developed.

BBCP has the potential to accomplish this task - and in significantly less time. The analysis completed at Kaspersky Lab, a world leader in malware reverse engineering, took more than two weeks to accomplish. Admittedly, BBCP is currently limited to only unpacked binary samples. However, after unpacking, an analysis similar to Kaspersky’s could be accomplished in mere minutes.

5.2. Interagency Communication

BBCP could be used in novel methods of interagency communication in the Intelligence Community (IC). As the IC now stands, separate agencies loathe to reveal many of their discoveries to other agencies, being rightfully fearful of their discoveries being broadcast to the entire world and foiling goals that they were pursuing. BBCP would enable the IC to share signatures of malicious code fragments to a common database that could associate actors, techniques and procedures. This database would augment the security posture of every organization with access to it, but would not divulge specific code, thereby avoiding any conflicts of interest between agencies. In the case of reverse engineering, analysts could check the database and request more information if their signatures were already present in the database, either based on an entire file or basic blocks, as BBCP enables. Agencies would be aware of the code that other agencies have already witnessed, but the submitting agency would have the option to not divulge any more information than what is required for defense, or what has already been submitted to the database.

5.3. Increased Efficacy of AV

Perhaps the most obvious application of this research is the vast improvements to current commercial anti-virus solutions it can provide. The BBCP algorithm is not difficult to implement and offers great improvements over BitShred. Decoupled from IDA Pro, commercial anti-virus could start doing simple and fast similarity analysis and effectively limit malware authors’ abilities to reuse code. Malware in response might migrate to memory only processes, but the inability to get on the hard drive would eliminate the threat of persistence.

If BBCP remained coupled with IDA Pro, then it would gain the ability to disassemble the plethora of processor architectures that IDA Pro natively understands. This includes Dalvik executable disassembly which would enable categorization of malware that runs on Android apps. This would almost be as highly desirable as eliminating persistence on personal computers. Android app marketplaces could run BBCP, eliminating any known malware and anything extremely similar to it. In the bring-your-own-device paradigm that the cyber realm is embracing, securing mobile endpoints like Android devices adds much value to an app marketplace.

5.4. Conclusions

Through careful examination of prior research, we able to quickly acquire algorithms that worked well at identifying shared code. We significantly increased the effectiveness of the algorithm that we found, mostly evidenced by the source subset test. BBCP avoids over-classification of similarity as evidenced by comparing dissimilar programs before we inserted the common netcat code.

BBCP does not just improve an algorithm, however, but also uses it in a novel way: the deep-dive dissection. As evidenced by the netcat experiment, DDD can be used to greatly speed up the reverse engineer’s task.

BBCP represents a heuristics-based detection technique that has been relatively unexplored in modern research. As thousands of new samples of malware are produced each week, many of which belong to previously-seen families, heuristics-based detection is becoming increasingly imperative to the fight. BBCP greatly improves upon previous research in the field of heuristics-based detection, and demonstrates a novel way to use a previously engineered algorithm. We also suggest multiple ways in which BBCP could be used to an even greater extent, such as provenance determination and
interagency communication. We also describe how BBCP would improve anti-virus solutions to the point that malware authors find persistence impossible. In the fight against malware, BBCP and other akin research endeavors raise the bar for malware authors, requiring them to rethink how they do “business.”

6. Acknowledgements

This work was created in the performance of a Cooperative Research and Development Agreement with the Department of the Air Force. The Government of the United States has certain rights to use this work. The Department of Homeland Security (DHS) sponsors the Center of Innovation at the United States Air Force Academy, which conducts research for educational purposes. The United States Air Force Academy and DHS sponsored the production of this material under United States Air Force Academy agreement number FA7000-11-2-0001. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of The United States Air Force Academy, the Department of Defense, or the U.S. Government.

7. References


