First Byte: Force-Based Clustering of Filtered Block N-Grams to Detect Code Reuse in Malicious Software

Jason Upchurch[1,2,3] and Xiaobo Zhou[3]
[1] Center of Innovation, United States Air Force Academy, CO, USA
jason.upchurch.ctr@usafa.edu
[2] Intel Labs, Intel Corporation, Hillsboro, OR, USA
jason.r.upchurch@intel.com
[3] Department of Computer Science, University of Colorado, Colorado Springs, USA
{jupchurc,xzhou}@uccs.edu

Abstract

Detecting code reuse in malicious software is complicated by the lack of source code. The same circumstance that makes code reuse detection in malicious software desirable, that is, the limited availability of original source code, also contributes to the difficulty of detecting code reuse. In this paper, we propose a method for detecting code reuse in software, specifically malicious software, that moves beyond the limitations of targeting variant detection (categorization of families). This method expands n-gram analysis to target basic blocks extracted from compiled code vice entire text sections. It also targets individual relationships between basic blocks found in localized code reuse, while preserving the ability to detect variants and families of variants found with generalized code reuse. We demonstrate the limitations of similarity calculated without first disassembling the instructions and show that our First Byte normalization gives dramatic improvements in detection of code reuse. To visualize results, our method proposes force-based clustering as a solution to rapidly detect relationships between compiled binaries and detect relationships without complex analysis. Our methods retain the previously demonstrated ability of n-gram analysis to detect variants, while adding the ability to detect code reuse in non-variant malware. We show that our proposed filtering method reduces the number of similarity calculations and highlights only meaningful relationships in our malware set.

Malicious software, or Malware, was produced at a rate of 9M samples a quarter in year 2012 [21]. This rate is significant and, while not yet proven, lends support to the widely held belief that malware is created with significant code reuse [15]. If this hypothesis is correct, the detection of code reuse in software could be used to detect new malware as well as to provide a mechanism to narrow authorship. Code reuse detection has applications outside malicious software to include intellectual property protection and vulnerability discovery [9][23][24].

Current malicious software protection hinges on detection through signature scanning. However, this method has serious limitations [27]. One of these limitations is the volume of signatures that must be created to counter the threat of malicious software. Code reuse detection provides a new solution to extend the signature method. First, there is a potential to reduce the number of signatures needed to detect the millions of malicious programs found in the wild. By examining inter-relationships found within malware, and only malware, code that is frequently used in malware can be targeted for signature creation to detect all of the malware to which it belongs. In addition, this technique would force malicious code writers to proactively vary their code and eliminate code reuse dramatically, slowing the production of new malware.

Another application of code reuse detection in malicious software is the ability to trace such reuse. The primary benefit of the ability to trace code reuse in malware is in classification of intrusion sets. The premise of this application is, code that is reused in a small portion of malware is likely to have been developed by the same group. This is a key component to attribution of malware to specific writers as well as an indicator for classification to a particular intrusion set. Such methods are already employed in industry; however, they are currently conducted in a much more manual fashion. The method proposed in this paper would provide a key tool to move away from manual analysis.

The study and analysis of malicious code is hampered by the amount of time necessary to analyze the code [5]. Methods to preventing duplicate analysis are in production [26]. However, these methods are limited to a specific tool used

in analysis or are not resilient to minor variations of code. The method proposed in our paper provides such a solution. As we can identify code reuse in compiled malware, we can also find the complement to code reuse detection, the detection of previously unseen code. Analysis of code segments could be conducted and applied to all malware to which it belongs and new code could be quickly identified for analysis.

Lastly, our proposed method is useful in vulnerability discovery. Code reuse, while speeding up the development cycle, also propagates vulnerabilities to other software projects. Our method, just as in work [15], allows for the detection of vulnerabilities in programs once the vulnerability is found elsewhere. Where work [15], as implemented, only detects the presence of code reuse. Our method detects both the presence and location of the vulnerability.

In this paper, our four major contributions are as follows:

1. Code reuse detection of highly similar, localized code within compiled binaries.
2. A simple normalization method to reduce the effects of hash waterfalling.
3. A force-based clustering visualization solution to navigate a many-to-many relationship map.
4. Filtering of commonly found code.

1 Related Research and Motivation

The automatic detection of code reuse in compiled binaries has applications in malware detection, malware attribution, intellectual property protection, and bug detection. Research work [15][24][11][8][25][18][17] has been geared toward detection of malware through this reuse. However, proposed methods [15][25][11][1][17] were based on the detection of variants (families). Proposed methods [8][4][18] were based on the detection of behavior. And work [24] doesn't address malware and relies on at least 40 consecutive instructions to detect similarity, which is below the average number of instructions per basic block. While these methods are useful, and some of the techniques are used in our research, they are limited in that they can only detect entire program (or multiple consecutive blocks) similarity and behavior. None of these methods are particularly well suited to detect actual code reuse as commonly found in development. To achieve the desired contributions in malware attribution, program executable sections are not the unit for experiment, but the components that comprise the logic within the program. While the detection of localized code reuse in [7] has shown to be very useful, its technique of direct byte comparison is limited in scale. To this end, we explore scalable similarity tests based on basic blocks extracted from program executable sections individually, rather that the section as a whole.

We also use set membership methods [15] as a filtering mechanism to reduce noise in our proposed method. As we propose to move away from simple set membership tests to a more complex many-to-many associative relationship graph, we explore visualization of such relationships through force based clustering. Finally, we recognize the computational difficulty of this On^2 association problem and we set forth ideas on how to reduce n to more manageable value through filters that are linear in nature.

During this research, static analysis of compiled code was examined. Our goal is to detect source code reuse in compiled executables. Therefore, the detection of behavior such as work [8] are not as valuable, as behavior is not indicative of implementation. This is particularly true of the system call method in work [2]. As such, we focus on methods where the source machine code is extracted directly, processed, and compared.

We examined methods proposed in works [15][25][6]. The methods proposed in works [25][6] have several advantages. First, the work [25] provides an intermediate language to reduce and possibly overcome compiler generated differences from exact duplicate source. It also provides an emulator to allow for unpacking, which is a hurdle that must be overcome to transition the research. Lastly, the work [6] allows for the extraction of entire algorithms for analysis, vice basic blocks. The comparison of entire algorithms is more closely aligned with code reuse detection than basic blocks. However, each of these methods represent an end goal of detecting code reuse even when opcodes are radically dissimilar. For this current work, we choose a hybrid method based on n-gram analysis [1] used in malware works [13][14][15], simple normalization to improve the accuracy of n-gram analysis, filters to reduce uninteresting comparisons, and force-based clustering to visualize similarity results beyond detecting malware families.

2 Code Reuse Detection and Visualization

The work presented in this paper comprises of: retargeting n-gram similarity analysis to the basic blocks extracted from malware samples, the straight forward normalization of basic blocks to improve similarity matching, the clustering of resulting similarity graphs using push-pull force based clustering, and the filtering of basic blocks found in known benign software.

2.1 Similarity

The use of n-gram analysis for feature detection in works [1][14][13][15] is attractive. Our choice of n-gram analysis was made as it provides most promising results and scalability when used for family detection [15]. The n-gram extraction and feature hashing allows for a quick reimplementation for exploration of both similarity and other objectives of this research. We first re-implement work [14] as described, except the target of comparison is basic blocks vice executable sections.

A test set of malware was created from our malware sample set obtained through Open Malware [22] by selecting only C and C++ compiled code, which had no evidence of being packed. While obfuscation through packing is an issue for any complete analysis system, our work is confined to detecting similarity in unobfuscated code and deobfustaction is outside of our scope. This list was generated from simple signature based analysis [16]. Cumulative results of basic block analysis are comparable to work [15] as one would expect because basic block analysis is simply a subset of executable section analysis.

Basic blocks are extracted through batch processing via IdaPython [10]. We use IdaPython to identify and access each basic block in the sample program. Normalization (below) is completed at this step and a preprocessed dataset is created, identifying all basic blocks found within the sample binary. This process generates an average of 281 (mean 91, max 6800 in the sample set) basic blocks, defined by control flow changes within the compiled binary of the malware, per sample. This number is consistent with self executable files (exe's) within Windows XP (dll's contain much of the code base for executables) and is expected due to the large number of small downloader executables within the malware base.

Similarity is calculated by sliding window feature hash across the extracted basic block. While it would be possible to store each hash, the resulting dataset would be very large and the computation would be cumbersome. We did not attempt this as it has already been proven in work [15] to be very computationally expensive. However, such storage would allow for similarity to be calculated via a Jaccard index by calculating the union and intersection of each basic block's feature hashes. Given two feature sets F_a, F_b extracted from basic blocks B_a, B_b, which in turn are extracted from malware M_a, M_b, similarity of the basic blocks is determined via the Jaccard index: $J(F_a, F_b) = \frac{(F_a \cap F_b)}{(F_a \cup F_b)}$. However, as noted in work [15], storage of each n-gram hash from each basic block is not optimal. Instead, the feature hash is stored as an index in a bit array, which creates a probabilistic data structure for which similarity can be calculated. Simple unions and intersections are easily, and more importantly, quickly calculated from such structures via simple bit operations. We use this circumstance to calculate an approximate Jaccard Index as in work [15]. That is, $J(F_a, F_b) \approx \frac{S(B_a \wedge B_b)}{S(B_a \vee B_b)}$ where the Jaccard Index $J(F_a, F_b)$ is approximated by taking the bit AND and the bit OR of each bit array B_i, counting the set bits with function $S()$, and dividing the result. It is an approximation due to the probability of collision within each bit array. However, unlike work [15], our work only examines one byte per instruction, guaranteeing the number of instructions per window to be constant. The is advantageous for several reasons. Representing instructions as a constant length does not allow arbitrary long instructions to increase the probability of match my consuming more of the window than other instructions. Long instructions consuming more window space than other instructions also flattens the structure of the sequence of instructions by reducing number of instructions per window. Representing instructions with only their first byte also reduces the number of windows examined per basic block; therefore, less windows are indexed into our bit array, reducing potential collisions.

2.2 Normalization

When using the raw instruction method in work [15], we found that many of the potential matches between functions were obscured by operand values, particularly in the case of memory addresses. This should be of no surprise based on works [24][25]; however, initial results without code normalization were more sparse than desired. An examination of sample assembly extracted from test code shows that fouling due to relative memory location changes is pronounced with even small modifications to code, such as variable insertion. However, such fouling is reduced significantly if destination addresses can be ignored. We refer to this normalization procedure as the First Byte method in the rest of this paper. Conversely, the n-gram consisting of the full instructions are referred to as the Full Byte method.

As we already used IDA [10] to extract basic blocks from each program, we instead extracted only the First Byte of each decoded instruction from the basic block. This not only removes some operands from specific instructions, particularly control flow instructions with relative offset destinations, but also preserves the structure of the basic block. In addition, the method greatly reduces the number of hashes calculated in the n-gram analysis and does not exclude basic blocks that don't match simply due to operand specifics. Figures 5 and 6 show the effect of normalization on compiled byte code.

2.3 Clustering

We desire to determine not just if a program belonged to a family, but identify what specific code in the malware

was related to other specific code. To achieve this goal, all relationships must be measured and relevant relationships recorded. Since each basic block extracted from a program must be compared to every other, this manifests On^2 complexity. In addition, initial experimentation shows a very large amount of relationships generated from small numbers of samples.

The preprocessed basic blocks are hashed via sliding window as in works [14] [15]. A graph is constructed with malware samples as parents and the sample corresponding basic blocks as children. Each basic block is compared to every other basic block and a similarity is calculated by dividing feature hash matches by possible matches. If the similarity score exceeds a prescribed threshold, an edge is created between the corresponding basic blocks. The edge weight is the similarity score.

Clustering is accomplished via method [3] with the imported algorithm [20]. The clustering is force-based rather than flat nearest neighbor [15]. This approach retains similarity connections between malware samples that are generally dissimilar. This produces a graph (Fig 1) that visually contributes understanding to the complex nature of the data. Not only are variants revealed, but each relationship is shown in the graph. The number of relationships can, however, be large in a relatively small number of malware samples. To reduce the number of relationships, we introduce filtering.

Figure 1: 96 malware samles, with 25k basic blocks, clustered, location of two variants.

2.4 Filtering

Filtering is accomplished by examining known goodware in a similar method to the examination of malware. The goodware undergoes the same processing as the sample malware. We first extract out the basic blocks from the executable $text$ section; however, we do not list each basic block for comparison. We instead store all feature hashes within a large bit field, creating a filter to test for set membership. Each basic block extracted from malware is then compared to the feature hashes of the filter. If the basic block is found to be a member of the filter, it is included in the graph for completeness. However, it is not compared to all basic blocks. Thus, this linear method reduces the number of similarity checks conducted given a malware set and reduces the complexity of the associated graph.

3 Experimental Results

3.1 Equipment and Dataset

All experiments were performed on an OS X machine (Intel 2.3 GHz i7 / 4 Core, 8 Thread / 16GB memory) using only a single core unless otherwise noted.

We performed our experiments on a malware data set supplied by the Open Malware Project [22]. Our test subset was generated by processing our dataset in order of md5 hash and identifying the first n samples that tested positive for C/C++ signatures via the PEiD [16] database. Our use of this selection process ensures that the samples are random within the domain of C/C++ malware. The Open Malware Project dataset consists of more than 5M malware samples by MD5 hash. We limit our analysis to C/C++, However, works, such as Juxapp [9], show that n-gram analysis is effective across other languages.

3.2 Similarity

Sample runs of 10k basic blocks were conducted to determine the effect of window size and similarity threshold on resulting similarity matches. Similarity was measured between 30-100% in 10% increments for both First Byte and Full Byte methods. Figure 2 shows the results of the effect of similarity thresholds on the number of matches recorded when the window size is held constant at 10 bytes in both the First Byte and Full Byte methods. In the First Byte method, each byte represents a single instruction, while the Full Byte method varies due to the x86 variable opcode sizing. The Full Byte method does not intelligently account for instruction size; however, x86 is confined to no more than 15 bytes per instruction [12]. Therefore, the n-grams of size 10 may not cover an entire instruction. Thus, First Byte analysis produces more matches while covering

more instructions. This, of course, comes at the cost of more possible collisions due to more than one instruction being represented by a single byte.

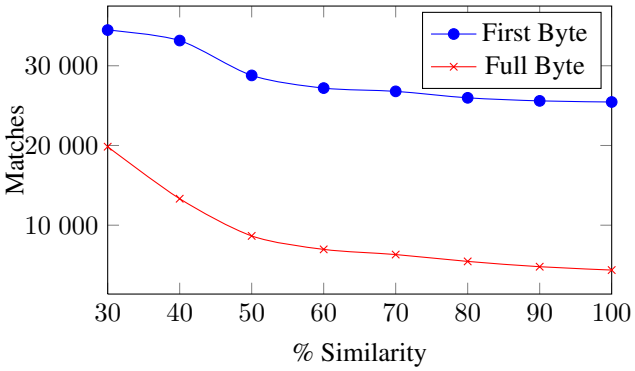

Figure 2: Similarity Threshold VS Matches Detected

Window size was measured between 5-500 bytes. Figure 4 shows the effect of window size on the number of matches when the similarity threshold is varied. As expected, as n-gram size increases, the number of matches decreases. However, the rate of decrease is higher with Full Byte n-grams verses First Byte. Despite the number of instructions included in First Byte n-grams being much higher and a reduction in the number of basic blocks analyzed, there is an increase in matching with First Byte compared to Full Byte. We believe this is due to the result of n-gram fouling (below) being more pronounced in the Full Byte analysis when compared with the First Byte.

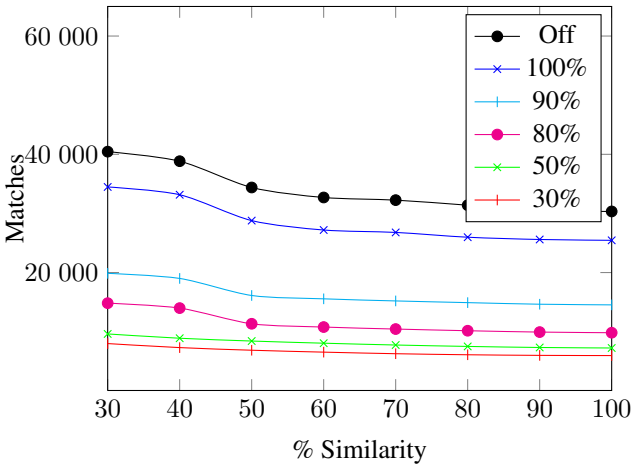

Figure 3: Effect of Filtering on Similarities

Without optimization, the On^2 nature of a many-to-many comparison is taxing. Our first run on unoptimized code, conducting all comparisons was 82 minutes for 25K basic blocks. However, by observing that basic blocks that differ drastically in size are not likely to be similar, we were

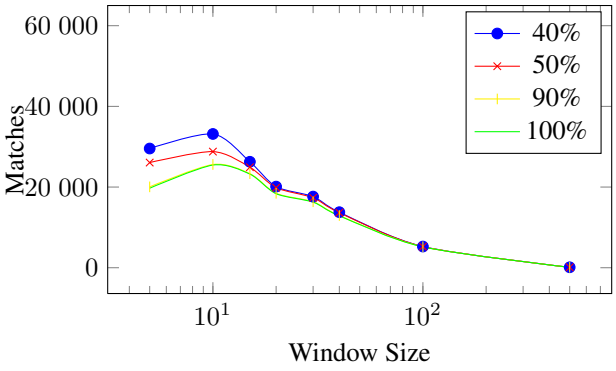

Figure 4: Window Size vs Matches Detected

able to reduce execution for 25K blocks to 4.5 minutes by bypassing blocks that were beyond a size threshold. In addition, filtering for know code (below), multi-threading, and superset comparisons (below), we can compare 25K blocks (3.8M instructions) in approximately 75 seconds.

3.3 Normalization

The effects of normalization can be seen in both the statistical analysis of numbers of matches found and the resulting graphs. Normalization in our method reduces each variable length instruction to a single byte. It reduces the overall basic block length and the number of sliding windows to compute the feature hash. Constrained at 25k basic blocks, our method discovered 2-5x more similarities (at reasonable thresholds) than the Full Byte method (Fig 2). Examination of both First and Full Byte verified the actual similarity of basic blocks and known variants (families) were still detected in our method.

Our First Byte method is much more resistant to hash waterfalling vice Full Byte. Figure 6 shows the effect of simple hash waterfalling with a single variable addition in the byte code of CRC.c. The modification adds a variable to the stack, changing the relative offsets. Hash waterfalling has the effect of fouling the feature hashing of each code segment. Figure 5 shows the effect of feature hashing using our First Byte method. Figure 6 shows the effect of waterfalling when using full instruction machine code. The fouling produced by variable insertion is much more pronounced in the Full Byte method verses our First Byte method as show in these examples. In the Full Byte method, it is reduced to a 31% match with the insertion of a single variable, while in the First Byte it is reduced to 73%.

The First Byte method is only affected by the addition of the `mov` instruction to move the variable onto the stack. The Full Byte method is also affected by the additional `mov` instruction; however, it is also affected by the change in rel-

```
55 89 83 C7 C7 EB 8B 03 8A 0F 89 8B 89
E8

55 89 83 C7 C7 C7 EB 8B 03 8A 0F 89
8B 89 E8
```

Figure 5: First Byte Waterfall Fouling, Window=5, 73% Match.

```
55 89 E5 83 EC 14 C7 45 FC 00 00 00 00
C7 45 F0 FF FF FF FF EB 52 8B 45 FC 03
45 08 8A 00 0F B6 C0 89 45 F4 8B 45 F4
89 04 24 E8 46 FF FF FF

55 89 E5 83 EC 24 C7 45 FC 00 00 00 00
C7 45 F4 00 00 00 00 C7 45 EC FF FF FF
FF EB 5A 8B 24 E8 3F 45 FC 03 45 08 8A
00 0F B6 C0 8B 45 F0 89 04 FF FF FF
```

Figure 6: Full Byte Waterfall Fouling, Window=5, 31% Match.

ative addressing caused by the insertion of that instruction. As the First Byte method is resistant to changes in most relative addressing, it matches more closely. As n-gram sizes increase, the effects of fouling become more pronounced.

3.4 Clustering

Detecting individual similarity leads to very large datasets even with relatively small input sizes. To tackle this problem, we borrow from force-based clustering methods developed in bioinformatics and network analysis. The initial map defines all nodes and edges; however, positions of nodes are not defined and the resulting network map is obscured in the complexity of unsorted information. Force-based clustering allows for visualization of complex network maps. In our research, we explore the OpenOrd [20] algorithm though [3]. This algorithm was chosen as it is known to produce good results with large graphs and is reasonably fast.

Our similarity comparison of 96 files with 25k basic blocks produces a graph file that is 182k lines long. The reason is that each node and edge must be recorded along with attribute information, such as edge weight, color, and label. This is a large amount of information and far too much to understand without additional analysis. It would, of course, be trivial to produce a look up system to examine one blocks relationship with other blocks; however, the extreme number of relationships provides a dataset that those attempting manual analysis would find daunting. To examine one sample's relationship with others is much more complex as each sample contains many basic blocks. Filtering would, of course, reduce the number of relationships by removing common, thus uninteresting, edges. However, in general, identifying code reuse is information intensive. Clustering techniques, even with small sample sets, revealed two variants (Fig. 1 –Artimis in upper left and Troj_gen R70H1HR in lower right), thus we preserve family detection as in [15][25][11][1] while adding localized detection small code reuse.

3.5 Filtering

The development of an accurate filter has two goals. First is to reduce the number of trivial relationships and the second is to reduce the number of On^2 complex comparisons. Filtering is accomplished much the same as methods [15] as specific identity need not be retained, only set membership. In this way, a large bit array is used to filter out trivial relationships. The filtering process is accomplished with linear computation time, reducing the On^2 load of the similarity comparisons. Filters are feature hash based and constructed with the same algorithm used in similarity, except that the filter does not track individual relationships, only membership. In this way, the filter is built in linear time and the comparisons are done in linear time. If membership is determined to be true, that basic block is not processed within the similarity engine.

Filter construction is based on known good-ware. To demonstrate the usefulness of filters, 100 random (non-obfuscated) binary executables were selected from our known clean examination system. A filter was generated from both the First and Full Byte basic block extraction methods over the same 10 byte window used in the base similarity experiment. The similarity model was run again with both the First Byte and the Full Byte methods, this time filtering out any basic block found in the filter set. A comparison of the reduction in can be found in Fig 3. Any filtered basic block is added to the graph for completeness; however, such basic blocks are not compared to other blocks for similarity. As such, the 25k block constraint includes filtered basic blocks. Examination of the unfiltered graph in Fig 7 and the filtered graph in Fig 8 show a dramatic reduction in complexity, while retaining relevant similarity from malware basic block n-gram analysis.

4 Our Key Contributions

4.1 Localized Similarity

Our n-gram analysis of basic blocks is successful in detecting code reuse in malware. It is straight forward to re-implement and resistant to minor code changes. Unlike other methods, our method determines localized similarity

Figure 7: 25k Blocks, Unfiltered, 50172 similarites found

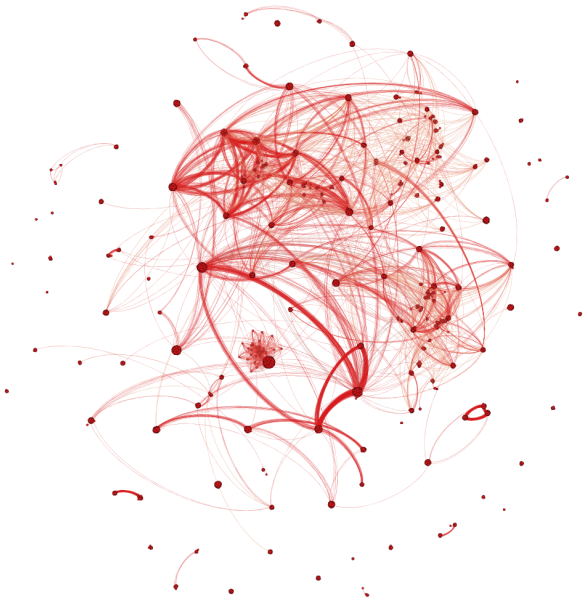

Figure 8: 25k Blocks, Filtered, 9828 similarities found

(small code reuse) and general similarity (family of variants) in a scalable solution. To allow our method to be more resistant to minor code changes, we also introduce a simple normalization method, vice more complex methods in [24][17].

4.2 Normalization

The nature of ASM is that there is little abstraction. This is particularly true in compiled machine language, and its ASM representation, as there is no use of labels for code references. Instead, hard offsets are coded into the machine and all labels are lost. As a result, control flow changes in the form of a call or jump have little chance of matching from one program to the next on the same compiler, when even the smallest change to the code is made. In addition, the choice of registers and the use of stack references by the compiler may change as well from program to program. All of this provides opportunities to miss similarity between basic blocks when using methods in work [14].

Our method intelligently examines the instruction itself and removes most operand specifics that would otherwise prevent identification by simply truncating the byte code of the instruction beyond the first byte. While there are certainly more precise methods to normalize byte code [25][24][19], our method is simple and very portable. It requires no complex algorithm or reference to be implemented. Given the time necessary to analyze the 200M malware samples in the wild, such a method has its advantages.

The benefits of this type of code normalization allow for more instructions to be included within each n-gram without significantly increasing the risk of a memory or register specific reference obscuring the similarity results. In addition, as each instruction is only represented by a single byte, each basic block is reduced in size. The resulting effect is that analysis time is reduced by decreasing the surface area of each basic block and increasing the area each n-gram can safely cover within the basic block. To demonstrate the effect, we produced maps based on full opcode analysis and compared the results with our First Byte analysis method (Fig 7 and Fig 8).

A seemingly obvious solution to our quadratic computational complexity is to apply locally-sensitive hashing (LSH) nearest neighbor techniques to find similarities as in work [24]. In Saebjornsen et al 2009, a solution to detecting code clones (code reuse, though larger than a single basic block must be evaluated) in software was put forward. The technique was similar in structure to our approach in that it disassembled the subject binaries, normalized the assembly, and then constructed windows to be compared. Their process, as with ours, use linear computational time to disassemble and normalize; however, they put forward a LSH based comparison algorithm to detect similarity. LSH is

known to be $O(n\ log\ n)$ and would seem to be a solution to scalability. However, looking at comparison times, Saebjornsen compares 5M instructions in 40 instruction windows, stride 10, in just over 200 minutes for a rate of 2500 window comparisons per minute (one core 2.66GHz Xeon). Our method compares 3.8M instructions in 14 minutes for a rate of 271K window comparisons per minute (one core 2.3GHz i7). Moving to 4 cores, out method compares 3.8M instructions in 4.5 minutes with 10 instruction windows, stride 1, for a rate of 844K window comparisons per minute. Computational time of comparable datasets and comparable computation via our method is 100x that of [24].

While it appears that the LSH method in work [24] is much slower with experimental sized datasets, there is little doubt that $O(n\ log\ n)$ LSH will outperform On^2 in transition given no limits on memory in which to store LSH buckets. However, the cost of moving to disk by exhausting memory is steep as is the cost of reduced accuracy when bounding LSH to memory. In Jang et al [15], they demonstrate that a map reduce based cluster of 64 worker nodes, each with 8 cores, 16 GB DRAM, 4 1TB disks and 10GbE, can process 1.9M samples per day. By extension, the same (or better performance given our smaller signatures) could be expected with our proposed work. This is well within reach of transition.

The sparse nature of our signatures and the low probability of match can be used to further reduce computation by combining sets of features into supersets. Each superset is then compared to other supersets or features individually. If the threshold for match is not met for the superset, it cannot be met by any of its member sets. Superset comparisons are limited by the increasing probability that false positives will overcome computational benefits; however, increasing the size of feature signatures reduces such probability. To test the benefits of this concept, we incorporated a 50 signature superset into our algorithm in which every candidate set was compared before conducting an individual comparison. This simple, single stage optimization resulted in a 3x increase in comparison speed.

4.3 Clustering

Force-based clustering allows both a macro view of similarity through clustered groups and a micro view of what relationships contributed to those groups. Unlike the work [15], our approach is not clustering for general family relationships, but looking for code reuse while maintaining the ability to detect families. Force-based clustering allows to achieve our goal of visualizing relationships of code reuse specifically without sacrificing variant detection. Fig 1 shows the effect of force-based clustering on both visualizing individual instances of code reuse and the detection of variants.

Our First Byte method calculates similarity of individual basic blocks prior to visualizing the results via force base clustering. The separation of visualization and similarity calculations allow for maps to be created that are generated with a focus in mind. While this paper focuses on the many-to-many relationships of a general pool of malware, it is also applicable to a detail one-to-many examination of malware. A simple change in focus could generate maps to produce 1, 2...n order relationships focused on a particular malware sample.

4.4 Filtering

Our work allows for the location and identification of code reuse within malware. The isolation of implemented functionality through basic block extraction allows for the comparison of that data to a known filter as well as other basic blocks. Code that is imported or inlined from widely available sources or the compiler itself is found in both malware and everyday software. We provide a method for excluding this code though a filter. This filtering greatly reduces noise in the resulting similarity maps by removing uninteresting code found in everyday software, which would be particularly well suited to finding provenance in attribution examinations.

5 Conclusion

In this paper, we have demonstrated that code reuse detection is possible within compiled code, while maintaining the ability to detect variant families. We have provided a normalization technique that reduces the fouling of n-gram feature hashing with respect to relative offset changes. Our use of basic block extraction also provides an avenue to exclude analysis of uninteresting similarities in malware by providing filters that can be used to exclude matches found in everyday software. Finally, we show that force-based clustering is a viable method to visualize complex results derived from basic block n-gram analysis of malware. In this work, we have provided an effective mechanism for code reuse detection in mass, but a method to visualize and cluster the relationships for that the data is meaningful without complex analysis. Future work will focus on examining improved algorithms and distributed clustering for scalability; focused and hierarchical graphing to manage the increased dataset size; and other methods of calculating similarity for increased detection.

Acknowledgment

The authors would like to thank the Open Malware Project for generously supplying the malware dataset for

this research project. 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 portions 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. Distribution Statement A: Approved for public release; distribution is unlimited. 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 or the U.S. Government.

References

[1] T. Abou-Assaleh, N. Cercone, V. Keselj, and R. Sweidan. N-gram-based detection of new malicious code. In *Computer Software and Applications Conference, 2004. COMPSAC 2004. Proceedings of the 28th Annual International*, volume 2, pages 41–42. IEEE, 2004.

[2] M. Alazab, S. Venkataraman, and P. Watters. Towards understanding malware behaviour by the extraction of api calls. In *Second Cybercrime and Trustworthy Computing Workshop*, pages 52–59, 2010.

[3] M. Bastian, S. Heymann, and M. Jacomy. Gephi: An open source software for exploring and manipulating networks, 2009.

[4] U. Bayer, P. M. Comparetti, C. Hlauschek, C. Kruegel, and E. Kirda. Scalable, behavior-based malware clustering. In *Network and Distributed System Security Symposium (NDSS)*. Citeseer, 2009.

[5] U. Bayer, C. Kruegel, and E. Kirda. Ttanalyze: A tool for analyzing malware. In *15th Annual Conference of the European Institute for Computer Antivirus Research (EICAR)*, 2006.

[6] U. Bayer, C. Kruegel, and E. Kirda. Anubis: Analyzing unknown binaries, 2009.

[7] P. M. Comparetti, G. Salvaneschi, E. Kirda, C. Kolbitsch, C. Kruegel, and S. Zanero. Identifying dormant functionality in malware programs.

[8] M. Fredrikson, S. Jha, M. Christodorescu, R. Sailer, and X. Yan. Synthesizing near-optimal malware specifications from suspicious behaviors. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 45 –60, may 2010.

[9] S. Hanna, L. Huang, E. Wu, S. Li, C. Chen, and D. Song. Juxtapp: a scalable system for detecting code reuse among android applications. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 62–81. Springer, 2013.

[10] Hex-Rays. Ida pro disassembler.

[11] X. Hu, T.-c. Chiueh, and K. G. Shin. Large-scale malware indexing using function-call graphs. In *Proceedings of the 16th ACM conference on Computer and communications security*, CCS '09, pages 611–620, New York, NY, USA, 2009. ACM.

[12] Intel. *Intel® 64 and IA-32 Architectures Software Developer's Manual Combined Volumes:1, 2A, 2B, 2C, 3A, 3B, and 3C*, 325462-046us edition, March 2013.

[13] J. Jang and D. Brumley. Bitshred: Fast, scalable code reuse detection in binary code (cmu-cylab-10-006). *CyLab*, page 28, 2009.

[14] J. Jang, D. Brumley, and S. Venkataraman. Bitshred: Fast, scalable malware triage. *Cylab, Carnegie Mellon University, Pittsburgh, PA, Technical Report CMU-Cylab-10-022*, 2010.

[15] J. Jang, D. Brumley, and S. Venkataraman. Bitshred: feature hashing malware for scalable triage and semantic analysis. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 309–320, New York, NY, USA, 2011. ACM.

[16] Q. Jibz, X. Snaker, and P. BOB. Peid. *Available in: http://www. peid. info/. Accessed in*, 21, 2011.

[17] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Polymorphic worm detection using structural information of executables. In *Recent Advances in Intrusion Detection*, pages 207–226. Springer, 2006.

[18] A. Lanzi, M. Sharif, and W. Lee. K-tracer: A system for extracting kernel malware behavior. In *Proceedings of the 16th Annual Network and Distributed System Security Symposium*, 2009.

[19] Z. Lin, Z. Lin, X. Zhang, and D. Xu. Automatic reverse engineering of data structures from binary execution. *In Proceedings of the 17th Annual Network and Distributed System Security Symposium (NDSS'10)*, 2010.

[20] S. Martin, W. Brown, R. Klavans, and K. Boyack. Openord: an open-source toolbox for large graph layout. In *IS&T/SPIE Electronic Imaging*, pages 786806–786806. International Society for Optics and Photonics, 2011.

[21] McAfee. First quarter thread report 2012, 2012.

[22] Open-Computing. Community malicious code research and analysis, 2009.

[23] K. Rieck, T. Holz, C. Willems, P. Düssel, and P. Laskov. Learning and classification of malware behavior. *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 108–125, 2008.

[24] A. Sæbjørnsen, J. Willcock, T. Panas, D. Quinlan, and Z. Su. Detecting code clones in binary executables. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, pages 117–128. ACM, 2009.

[25] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. Bitblaze: A new approach to computer security via binary analysis. *Information systems security*, pages 1–25, 2008.

[26] G. Tech. Titan case study, 2012.

[27] W. Yan and E. Wu. Toward automatic discovery of malware signature for anti-virus cloud computing. *Complex Sciences*, pages 724–728, 2009.