FuncTracker: Discovering Shared Code to Aid Malware Forensics Extended Abstract

Charles LeDoux, Arun Lakhotia, Craig Miles, Vivek Notani
charles@charlesledoux.com, arun@louisiana.edu, craig@craigmil.es, vivek200690@gmail.com
University of Louisiana at Lafayette

Avi Pfeffer
apfeffer@cra.com
Charles River Analytics

Abstract

Malware code has forensic value, as evident from recent studies drawing relationships between creators of Duqu and Stuxnet through similarity of their code. We present FuncTracker, a system developed on top of Palantir, to discover, visualize, and explore relationships between malware code, with the intent of drawing connections over very large corpi of malware – millions of binaries consiting of terrabytes of data. To address such scale we forego the classic data-mining methods requiring pairwise comparison of feature vectors, and instead represent a malware as a set of hashes over carefully selected features. To ensure that a hash match implies a strong match we represent individual functions using hashes of semantic features, in lieu of syntact features commonly used in the literature. A graph representing a collection of malware is formed by function hashes representing nodes, making it possible to explore the collection using classic graph operations supported by Palantir. By annotating the nodes with additional information, such as the location and time where the malware was discovered, one can use the relationship within malware to make connections between otherwise unrelated clues.

1 Introduction

Malware research is currently undergoing a shift in focus. With the rise of Advanced Persistent Threats (APT), it is no longer sufficient to simply know *what* happened. There is a growing need to understand the *who* and the *why* as well. Essential to this new focus of malware research is the discovery of relationships between various pieces of malware. Every attack, every instance of malware provides a tiny piece of the larger puzzle. It is only when these pieces are put together that the important questions can be answered. Relationships between malware help the researcher and analyst determine where each piece of the puzzle fits.

One of the key indicators of the existence of relationships is shared code. Code share between two malware can imply shared authorship, behaviors, purpose, lineage, tool chains, etc. For example, shared code was a major contributor in a report by Kaspersky [5] that postulated close ties between the authors of Duqu and Stuxnet. Malware forensics, then, would greatly benefit from a systematic way of discovering, encoding, and searching these relationships based on shared code.

There have been efforts in clustering related malware, but the application of these efforts in the general case of shared code is limited. Clustering algorithms are generally inefficient, taking on the order of $O(n^2)$. This can be an expensive prospect when dealing with millions of malware; it is an unaffordable cost when these millions of malware are each broken into hundreds or thousands of pieces. In addition, these clustering applications generally focus on the *whole* binary. Binaries that share only a *single* function will have very low similarity between them, and a single shared function will be missed.

We present FuncTracker, a system for discovering, exploring, and visualizing relationships between malware. Relationships are determined based on shared functions. This level of comparison was shown to provide a reasonable level of granularity by Cohen et al. [3] and Jin et al. [7]. Functions are determined to be similar based on matching semantic hash. These semantic hashes are computed using features at multiple levels of abstraction, both syntactic and semantic, designed to be robust against common obfuscations. The use of hashes instead of a similarity function, such as Jaccard Index, allows for scalability as the comparison is constant time.

The discovered relationships between malware are represented in a graph with annotated edges and nodes, thus allowing for efficient searching and navigation. In addition to shared functions, this graph based representation allows for any type of additional relationship to be represented, such as URLs contacted, APIs called, etc. Different types of searches are enabled by either travers-

```
mov   eax,0x5
add   ebx,0x4
imul  eax,ebx
```

(a) Code

```
mov   A, N1
add   B, N2
imul  A, B
```

(b) GenCode

```
eax = 5
ebx = def(ebx) x 5 + 20
```

(c) Semantics

```
A = N1
B = def(B) x N1 + N2
```

(d) GenSemantics

Figure 1: Features provided by BinJuice

ing the graph or by searching for nodes with a specific or matching annotations.

2 Discovering Shared Functions

Our central problem in FuncTracker is locating small, but non-trivial pieces of shared code in otherwise unrelated binaries. Computing whole binary similarity is too coarse as we expect the overall similarity to be low. Using a block level is too fine grain as it is likely there will be a large number of small, trivial pieces of code matching causing much noise. Functions, however, should provide a nice middle ground.

Functions are represented by the set of blocks (from the control flow graph) they contain. Blocks are represented by four features created using BinJuice [10]: Code, Semantics, GenSemantics (termed "Juice"), and GenCode. Each of these features is a single string representing a different layer of abstraction. An illustrative example is given in Figure 1.

Code is, as the name implies, the disassembled code of the function. This includes the actual instruction (its mnemonic) and the operands. A common abstraction from code is to use only the instruction mnemonics and not the operands [8, 11, 14]. However, by ignoring operands any data flow information is lost. BinJuice defines the feature GenCode that abstracts away from Code any specific registers and memory locations, using logic variable to capture the data flow relationships. The abstraction is performed such that two Code features that differ only in the names of registers and addresses used will map to the same GenCode feature.

The two syntax based features described above allow for strong matches, but they are also easy to break. Simply reordering instructions or substituting one instruction for a semantically equivalent one will result in a different Code and GenCode feature. Thus, BinJuice provides the Semantics feature using symbolic interpretation and algebraic simplification to represent the operational semantics of a block. The Semantics feature is designed to be the same for two blocks that are functionally equivalent, regardless of what the code looks like. GenSemantics is analogous to GenCode in both motivation and creation. Both Semantics and GenSemantics have a canonical form that allows for constant time comparison.

A major concern in searching for similar code is scale. Malware collections often number in the millions of binaries and each individual malware additionally contains many functions, multiplying the required scale by a hundred or thousand fold. In order to efficiently handle this scale of comparisons, we use hashes instead of classic distance functions. Four different kinds of hashes are created, one corresponding to each type of BinJuice feature. The hashes are created by sorting the features representing the blocks in the function, concatenating them, and taking a cryptographic hash. Lest this method appear naïve, work by Cohen et al. [3] showed that even a hash of raw bytes of a function, abstracted simply by zero-ing bytes representing memory addresses, is very effecting in finding similar code. Using hashes of semantics ought to provide a larger set of functions allowing a constant time lookup of similar functions.

3 Exploring Relationships

Discovered relationships must still be presented to an analyst in a usable and comprehensible format. We represent these relationships using a graph structure. Nodes in the graph represent objects such as a binary or a function; edges indicate a relationship between objects. Properties can additionally be attached to a node. Such a representation allows for intuitive visualizations and an easy method of constructing queries to explore the relationships. Searches can be performed by either querying for nodes with similar properties or by traversing the graph.

In our selected graph structure, the permitted types of nodes are binaries, functions, and blocks. A binary contains (has edges between) functions and a function contains blocks. Blocks have their respective BinJuice features attached as properties and functions have the various hashes as properties. An example query is to start from a binary of interest, traverse the graph to find all contained functions, query for functions with similar GenCode hash, and then traverse from the returned nodes to binaries. This results in returning binaries that share some function based on GenCode hash, along with the matching functions.

Graph representation offers the additional benefit of extensibility. The graph may be populated with additional types of nodes, edges, and properties; and all of the connections explored using graph traversal algorithms. For example, if known, we may associate binaries with their authors. We can then identify binaries likely written by the same author by traversing connections through similar functions. Similarly, if a set of functions in a particular binary are known to implement a certain behavior, this information may be associated with other functions with similar semantics.

4 FuncTracker

We have built an initial prototype titled FuncTracker on top of the intelligence platform Palantir[1]. New binaries are first disassembled and split into functions using IDA Pro[2]. The resulting functions are then passed to BinJuice [10], ignoring functions that IDA has marked as library code. FuncTracker next computes the hashes as described in Section 2. The graph structure described in Section 3 is created and loaded into Palantir. The built in capabilities of Palantir can then be used to explore the newly discovered relationships, draw additional connections, and collaborate with other analysts.

5 Example Use Case

We now illustrate the utility of FuncTracker with a controlled study using lab generated malware binaries containing known ground truth. Our study is based around our motivating use case: finding binaries containing functions implementing a known behavior. Starting from a single known implementation of a specific behavior, we locate binaries containing this implementation. This could be used, for example, to find binaries containing a behavioral implementation associated to known authors, thus associating the binaries with these authors.

The full dataset was first loaded into Palantir as described in Section 4. We then located two procedures known to implement a specific behavior and performed the following two queries in sequence: a) search for functions with the same GenSemantics as the two known procedures, b) find binaries connected to the functions returned by the previous query.

Figure 2 presents the result of the Palantir queries along with the ground truth. The two procedures that started the queries are shown at the top of the hierarchy. Below these are procedures that share the same GenSemantics as at least one of them. The procedure annotated in a blue box is a false positive. Below the procedures are the binaries that contain them and thus contain the behavior of interest. At the bottom, colored red, are the false negatives. There were 18 procedures and 2 binaries incorrectly returned.

As can be seen from this example, FuncTracker has the potential to establish reliable relationships between malware based on shared code. Only one false positive was returned, implying that matches based on the hash have a higher probability of correctness. This is important for many forensics tasks, such as assigning authorship to the malware. Even with the high degree of correct matches, only two binaries were failed to be retrieved. A number of matching procedures were missed, yet the relationships between malware were still established.

6 Related Work

BinDiff [14] uses a number of matching algorithms to discover similarities between two binaries and a heuristic to select the correct one. These algorithms are based on either the structure of the program (ex. call graph) or syntactic features (ex. instruction mnemonics, string references). Also relying on syntactic features, Rendevous [8] uses a classic “index and query” approach to find similar binaries. BinHunt [4] and Christodorescu et al. [2] use symbolic interpretation to determine the semantics of blocks, but their matching algorithms do not scale well as they attempt to brute force a matching of values (such as register names) to logic variables (such as in GenCode).

Kruegel et al. [9] defines a method of finding similar binaries by finding subgraph isomorphism between their colored control flow graphs. The coloring is done based on the general “class” of instructions present in a block. This method is used to good effect by BitShred [6] to group together both similar malware and the features causing them to be similar. This method, however, is useful only when the binaries being compared are expected to be similar.

Jin et al. [7] also match functions using hashes analogous to those we compute using GenSemantics. They generalize the semantics by using hashes of values computed by a basic block on a set of predefined, though random, inputs. Assuming a sufficiently large set of inputs, the hashes they compute would create similar partitions of basic blocks as those computed using GenSemantics.

7 Future Work

While our initial case study is encouraging, a comprehensive evaluation is still required. To this end, we are setting up an experiment designed to test the efficiency and accuracy of FuncTracker in retrieving all shared functions in a group of binaries. Additionally, we are looking at extending our hash based comparison. While it appears our current hash can be useful, many practical applications would benefit from the ability to use an approximate hash such as Locality Sensitive Hashing [1, 7, 13] or a Bloom filter based approach [6, 12].

8 Acknowledgments

This work was supported by Air Force Research Laboratory, Rome, NY and the Defense Advanced Research Projects Agency under contract FA8750-10-C-0171, with thanks to Mr. Timothy Fraser. The views

[1] https://www.palantir.com

[2] https://www.hex-rays.com/products/ida

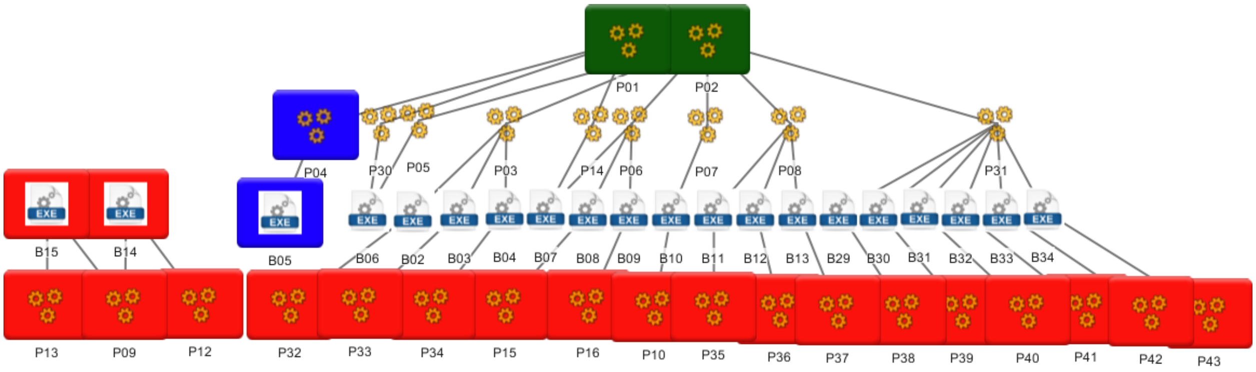

Figure 2: Using FuncTracker to discover binaries implementing a known behavior.

expressed are those of the authors and do not reflect the official policy or position of the US Department of Defense or the U.S. Government.

References

[1] U. Bayer, P. M. Comparetti, C. Hlauschek, C. Kruegel, and E. Kirda. Scalable, behavior-based malware clustering. In *16th Annual Network & Distributed Security Symposium (NDSS'2009)*, volume 9, pages 8–11. Internet Society, 2009.

[2] M. Christodorescu and S. Jha. Static analysis of executables to detect malicious patterns. In *Proc. of the 12th USENIX Security Symposium*, 2003.

[3] C. Cohen and J. S. Havrilla. Function hashing for malicious code analysis. In *CERT Research Annual Report*, pages 26–29. Software Engineering Institute, Carnegie Mellon University, 2009.

[4] D. Gao, M. K. Reiter, and D. Song. Binhunt: Automatically finding semantic differences in binary programs. In *Information and Communications Security*, number 5308 in Lecture Notes in Computer Science, pages 238–255. Springer, 2008.

[5] A. Gostev and I. Soumenkov. Stuxnet/Duqu: the evolution of drivers. https://www.securelist.com/en/analysis/204792208/Stuxnet_Duqu_The_Evolution_of_Drivers, Dec. 2011.

[6] J. Jang, D. Brumley, and S. Venkataraman. Bitshred: Feature hashing malware for scalable triage and semantic analysis. In *Proc. of the 18th ACM Conf. on Computer and Communications Security (CCS)*, pages 309–320. ACM, 2011.

[7] W. Jin, S. Chaki, C. Cohen, A. Gurfinkel, J. Havrilla, C. Hines, and P. Narasimhan. Binary function clustering using semantic hashes. In *11th International Conf. on Machine Learning and Applications (ICMLA)*, volume 1, pages 386–391, 2012.

[8] W. M. Khoo, A. Mycroft, and R. Anderson. Rendezvous: A search engine for binary code. In *Proc. of the 10th Working Conf. on Mining Software Repositories (MSR)*, pages 329–338. IEEE Press, 2013.

[9] 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*, number 3858 in Lecture Notes in Computer Science, pages 207–226. Springer, 2006.

[10] A. Lakhotia, M. D. Preda, and R. Giacobazzi. Fast location of similar code fragments using semantic 'juice'. In *Proc. of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop (PPREW)*. ACM, 2013.

[11] A. Lakhotia, A. Walenstein, C. Miles, and A. Singh. Vilo: A rapid learning nearest-neighbor classifier for malware triage. *Journal of Computer Virology and Hacking Techniques*, pages 1–15, Mar. 2013.

[12] V. Roussev. Data fingerprinting with similarity digests. In *Advances in Digital Forensics VI*, pages 207–226. Springer, 2010.

[13] A. Saebjornsen, J. Willcock, T. Panas, D. Quinlan, and Z. Su. Detecting code clones in binary executables. In *Proc. of the 18th International Symposium on Software Testing and Analysis (ISSTA)*, pages 117–128. ACM, 2009.

[14] Zynamics. BinDiff 3.2 manual. http://www.zynamics.com/bindiff/manual/.