2016 IEEE Security and Privacy Workshops

A Biosequence-based Approach to Software Characterization

Christopher S. Oehmen, Elena S. Peterson, Aaron R. Phillips, Darren S. Curtis
Pacific Northwest National Laboratory
Computational and Statistical Analytics Division
Richland, WA USA
{Christopher.Oehmen, Elena.Peterson, Aaron.Phillips, Darren.Curtis}@pnnl.gov

***Abstract*— For many applications, it is desirable to have a process for recognizing when software binaries are closely related without relying on them to be identical or have identical segments. But doing so in a dynamic environment is a nontrivial task because most approaches to software similarity require extensive and time-consuming analysis of a binary, or they fail to recognize executables that are similar but not identical. Presented herein is a novel biosequence-based method for quantifying similarity of executable binaries. Using this method, we show in an example application on large-scale multi-author codes that 1) the biosequence-based method has a statistical performance in recognizing and distinguishing between a collection of real-world high performance computing applications better than 90% of ideal; and 2) an example of using family-tree analysis to tune identification for a code subfamily can achieve better than 99% of ideal performance.**

Keywords—software analysis, sequence analysis, cyber security

I. Introduction

The organic variation in the population of binaries motivates an approach for recognizing "families and variants" of software binaries as opposed to "individuals". The ability to recognize related but distinct variants is essential for several practical applications, such as maintaining a large repository of software (such as the NIST repository) in which one would not want many redundant copies of closely related binaries, detecting the presence of freeware or other licensed code within a developing codebase, and ensuring that only certain applications are used in restricted environments (whitelisting), to name a few. For such applications, instead of traditional code analysis, which can require extensive computational power and far more detailed than necessary, there is a need to draw from techniques in other domains such as biological sciences that are more amenable to recognizing families and variants.

In this paper, a novel bio-inspired method for recognizing similar software is presented. This method is demonstrated to be a suitable algorithm core for the problem of executable binary whitelisting via verifying software identity against known examples of the software. In active environments, many slight variants of software may exist. It is unreasonable to expect that each executing individual is an exact match to any previously reported software individual. So biosequence analysis is an ideal technique to quantify the extent to which the new software matches a sequence-based model of software.

For bio-inspired software identity verification, a collection of software variants from the same code family is thought of as a "family" and each time a user wants to execute a code, that "individual" is checked to make sure it is truly a member of the family. As a simple example, the linear algebra application LINPACK is a different family than the computational chemistry code ADF. LINPACK versions 1.2 and 2.3 are different individuals within the same family. If a user of a multi-user platform wanted to run a modified version of LINPACK using a batch submit script, the process of software identity verification would analyze the binary before allowing the job to run to ensure that the executable being submitted is, in fact, a member of the LINPACK family by comparing it to a model of LINPACK family members. This approach to family analysis is a special case of a larger field of general software analysis, which has many different algorithmic approaches.

A. Existing Approaches to Software Analysis

Clone detection is an existing software analysis approach that could potentially be used to recognize highly similar variants of a binary family. Applications of clone detection are generally applied to large-scale code base software for the purpose of 1) finding and eliminating cut-and-paste segments in a large software projects because these are especially prone to introduce complexity and bugs, 2) identifying instances of software plagiarism, or 3) for making sure licensed code is free of open source code fragments or other software that would jeopardize a commercial license.

Clone detection is typically done either by analyzing source code, or by operating on the disassembled binary (*i.e.* the assembly instructions). With our method, the emphasis is on the binary itself, because detailed analysis of source code is unnecessary for the purpose of identifying similar binaries.

Work presented herein was partly funded by the Laboratory Directed Research and Development (LDRD) at Pacific Northwest National Laboratory (PNNL) via the Data Intensive Computing Initiative (DICI), Information and Infrastructure Integrity Initiative (I4), and Signature Discovery Initiative (SDI). Pacific Northwest National Laboratory PNNL is operated for the DOE by Battelle under contract DE-AC06-76RLO-1830.

© 2016, Christopher S. Oehmen. Under license to IEEE.
DOI 10.1109/SPW.2016.43

IEEE computer society

Authorized licensed use limited to: IEEE Xplore. Downloaded on October 19,2024 at 16:56:54 UTC from IEEE Xplore. Restrictions apply.

Also, for speed and efficiency reasons, linear transformations of executable binaries are preferred. The goal of the current work is to *not* rely on higher dimensional information that captures relationships between code segments within a binary to discover binary similarity.

Detecting similar binaries directly is the target of many commercial offerings and research projects, most of which are focused on malware detection, and are based on code signatures. Typically these signatures are built from checksums or other transformations of the binary sequence into numerical representations where finding a match is equivalent to finding an equal checksum. There are many variations on this theme including simplistic approaches where checksums are calculated for binaries [1] or sections of binaries like functions or functional blocks. Such exact-matching methods are not suitable for recognizing binaries in a dynamic environment, as the binaries should not be exact matches. Exact-match based approaches fail because even small modifications of code results in an entirely unrelated checksum value. To mitigate this issue some have used a locality-preserving hash value [2], but such methods presume one has already transformed raw data into a feature vector representation. The goal for our research is to operate on the binaries themselves.

Some approaches use sequences of disassembled instructions to identify clones [3, 4]. In our approach, instructions are resolved and exact matches are made more flexible by allowing insertions or mismatches that occur under a threshold. In more sophisticated approaches, disassembly is used as a preprocessing step to identify code segments such as functions that generate check-sum signatures [5-7]. This is a very promising approach, and has been shown to eliminate many duplicates or near duplicates from a corpus of known malicious software. But these methods are not readily available and their performance for whitelisting diverse families of large binaries is unreported. Other variants use techniques like using disassembly to identify the entry point of a binary as the starting point for an assembly stream signature. The main limitation of these approaches is the static nature of the signature itself. Though signatures do allow for some flexibility in binary recognition, a method is needed to "discover" the most reliable indicators of particular code families.

Taking this one step further, it has been shown that highly similar regions of a binary can be identified using disassembled, normalized sequences combined with locality sensitive hashing (a relaxed form of locality preserving hashing) [8]. Including dynamic analysis with normalized instruction sequences has been shown to increase sensitivity when program behaviors are correlated with static analysis [9], but dynamic analysis is unnecessary and too costly to perform on a regular basis.

Herein, two methods are introduced: *instruction frequency vector* and *bioinformatics-based* similarity analysis. These are two alternative approaches for software identity verification that are demonstrated to surpass the limitation of hash-based approaches (which would trivially fail to identify any similar but non-identical software artifacts). The bioinformatics-based approach has been demonstrated as a powerful tool when operating on Abstract Syntax Trees (AST) [10, 11]. The method presented uses a similar approach, but instead of using the AST, it only relies on disassembly of a binary. These methods do not rely on presence of source code, nor do they analyze complex lexical features or structures within the binaries and so operate on raw binaries quickly. They rely only on analysis of disassembled instructions. The motivation and features of these two approaches are described in greater detail in the following sections.

B. *Instruction Frequency Vector Similarity*

Instruction frequency vector-based similarity analysis tests the degree to which a global, frequency-based representation of a software binary's disassembled instructions can detect similar software instances, even when those binaries are non-identical. Vectorization is a "global" approach in that a single feature vector describes an entire code instance and therefore would be most useful in identifying when one entire instance is similar to another entire instance. This method would not be useful in detecting the similarity of individual parts. For the application of software identification verification, this whole code approach may be reasonable in many cases because one seeks to answer the question "is there enough evidence that a test binary is indeed a member of a predetermined binary family?" Global binary similarity methods such as this are not expected to work as well in large or highly variant families or subfamilies, but represent a simplistic starting point for rapid software identification verification and is included for comparison. Feature vector approaches make a simplifying assumption that the order of instructions is not important in distinguishing between binary families. As a consequence, feature vector-based approaches are computationally simpler than more complex analysis (such as bioinformatics-based), but are not expected to have the same statistical performance in identifying members of highly variant binary families.

C. *Bioinformatics-based Similarity*

Biosequence analysis provides an alternative to the limitations of hash-based and graph-based methods for binary analysis. Biosequences are chemical chains from a fixed number of subunits (4 subunit types for DNA and RNA, and 20 subunit types for proteins). Since DNA is inherited and modified from one generation to the next, similarities in DNA sequence (or in protein sequence which is related to DNA sequence) appear in organisms that share ancestors. Computational methods for discovering sequence similarity in biosequences have been developed and refined for decades [12, 13]. These methods are variations on dynamic programming approaches to map strings that represent biomolecules onto one another—a process called *alignment*. Alignment also results in a similarity score that can be used as a proximity metric.

Authorized licensed use limited to: IEEE Xplore. Downloaded on October 19,2024 at 16:56:54 UTC from IEEE Xplore. Restrictions apply.

The bioinformatics-based method presented here is based on the computational algorithm BLAST [14, 15], which is a statistical method for comparing text strings that represent biological chemical subunits. In BLAST, performing string alignment finds regions of commonality that exceed a statistical expectation, and is highly tolerant to mismatches, insertions and deletions. BLAST applied to binary analysis is an attractive alternative to hash-based methods because it tolerates a high degree of dissimilarity and it also has the ability to identify sub-regions of binaries that are highly similar. String matching in BLAST is more flexible than regular expressions, edit distance, and other traditional computational methods. BLAST compares a test string against a collection of reference strings to quantify the extent to which the test string is related to any of the reference strings. This is reported as a score as well as a statistical confidence measure for each test/reference pair having a score that beats a user-defined threshold. The calculated alignment between the strings is also reported. Fig. 1 illustrates how an alignment between two strings is reported in BLAST.

```
>Converted Code 1: Test sequence
PLYFRSADNFIOANEQKWINWAKLNFWAIFNCAOWLNK

>Converted Code A: Reference sequence
LOMKRSNFIOANELWWINAKLNFJRWAIFNCAOWLNKE

ALIGNMENT REGION:
RSADNFIOANEQKWINWAKLNF--WAIFNCAOWLNKE
RS--NFIOANELWWIN-AKLNFJRWAIFNCAOWLNKE
```

Figure 1. Alignment of protein representations of software showing (top panels) string representations of two code individuals and (bottom) the alignment between them. Dashes indicate places where one code contains inserted instructions not found in the other code. Bold characters denote mismatches between the individuals that are in the alignment region.

Because BLAST does not operate on higher-level structure, it does not incur the overhead of determining abstract syntax tree (AST) or program dependence graph (PDG) information from a binary, but has the potential to be more specific than frequency-based feature vector representations of code because key patterns in the sequence of instructions are preserved.

There are two main phases in this approach. In the first phase, members of a software family are disassembled, converted to sequences and analyzed using a high performance implementation of sequence analysis software. This newly developed tool called MADBlast, is a multiprocessor implementation of the BLAST sequence alignment method and was developed by the authors for related research. MADBlast allows for a more generalized alphabet than standard BLAST, and more efficiently utilizes nodes during task execution. In the second phase, a model of a code class (software family) is constructed using the results from the first phase for the purposes of comparing to new binaries. For simplicity, trivial family models are used, comprising a single code individual. Even this simplistic model achieves statistically significant results in identifying other members of the family, but for other applications the family model could be more sophisticated, further improving performance. Comparing a submitted executable to a library of acceptable code models in this second phase would not require HPC, but would still execute very rapidly.

II. Methods

A. Converting Executable Binaries and Normalization

Converting binaries to string representation begins with disassembly. In general the process of disassembly converts an executable binary into functional blocks of assembly instructions with their associated arguments. For example, a single addition operation in source code would be converted to a pair of "mov" operations to retrieve values from memory and place them in an arithmetic unit, a second operation to "add" the values, and another "mov" operation to place the result of the add operation into a new location in memory.

The disassembly method used here is based on a GNU GPL licensed project named Objconv that is distributed with most Linux operating systems. The disassembly produced by Objconv is of very high quality and comparable to that of IDA Pro [16], considered by many to be the *de facto* disassembly tool. Several other applications for disassembly including GNU Binutils Objdump utility, Ida Pro and ROSE [17] were also evaluated. In the case of Ida Pro and ROSE, sophisticated disassembly is achievable, but at a higher computational cost (in both memory and time to solution). Both applications maintain detailed information about many aspects of binary structure and functional layout that are not necessary for high-level characterization. Since the goal of sequence analysis is to quickly assess when binaries are more like each other than what one would expect by chance, data flow and other information is not necessary. On the other hand, the Unix utility Objdump was very fast but produced inaccurate disassembly of PE (Windows binaries) in some cases. Objconv provides an excellent balance of accurate disassembly and speed.

To automate the disassembly process into creation of files properly formatted for sequence analysis, Objconv was extended with a custom application called Disfast, which is a simple wrapper in either C++ or Python that allows users to control inputs to and process outputs from Objconv. Disfast also provides the conversion of Objconv output to a protein representation using the tokenization mentioned above. This format is called FASTA format (the format to which BLAST analysis can be applied), eliminating the need to post process files for alignment analysis.

Two types of normalization occurred to develop the mapping from raw binaries to frequency vector or biosequence representations. The first normalization step was to discard all arguments to the assembly instructions. The second normalization step filtered out both highly frequent and very

Authorized licensed use limited to: IEEE Xplore. Downloaded on October 19,2024 at 16:56:54 UTC from IEEE Xplore. Restrictions apply.

infrequently occurring instructions. The top frequently occurring instructions were ignored as they occur so frequently they would dominate any representation of the binaries. This resulted in 27 groups of instructions that covered the majority of instructions in the corpus by occurrence. All of the remaining less-frequently occurring instructions were ignored. This filtering step is done to maximize the information content of both the vector and biosequence binary forms by not reserving space or characters for highly infrequent instructions.

B. *Instruction Frequency Vector Method*

Each normalized binary sequence was represented as a feature vector having 27 elements, the value of each element being the relative abundance of a particular instruction in the binary. This obscured any effect that length of the binary might have. To generate a distance measure between the ith and j^{th} binaries, the Euclidean distance (D_{ij}) between their frequency vectors was calculated. Since the vectors were normalized to the positive hypersphere, a simple similarity measure of $1 - D_{ij}$ was calculated between all pairs of binaries.

C. *Bioinformatics-based Similarity Method*

Applying biological sequence analysis to software binaries is a several step process. First, instruction sequences were filtered as described above and binaries were transformed into the protein representation to be used by the BLAST algorithm implemented in the MADBlast tool. MADBlast was used to perform sequence alignment to find related subsequences. MADBlast takes as input text strings, a scoring matrix that contains reward values for text alignment and misalignment events, and scoring parameters such as the gap opening and gap extension penalties. This method has been generalized in prior work to be applicable to string alphabets beyond just the standard 20 amino acid characters expected by biological BLAST codes. A scoring matrix was also developed that is specific for binary analysis. Some of the details about refactoring this code are included here, but for a more complete description, please see [18]. These steps are described in more detail in the following sections.

1) *Generating Similarity Scores for Sequences*

MADBlast was used to compare each of the sequences disassembled from a corpus of HPC binaries to each of the other sequences from this corpus. This produced a BLAST output file with a record of the sequences that significantly aligned with each functional block from each binary with all of the functions from all of the other binaries in the corpus. The resulting MADBlast scores were the basis for using individual sequences as binary models to discover other similar binaries. These scores were also used as distance measures for family tree analysis using the hierarchical clustering tool WEKA [19] with Euclidean distance as the distance measure.

Two different stringency levels were tested for a positive “hit” between sequences. A less stringent cutoff was defined by sequences sharing an alignment of at least 10 characters with at least 50% identity (Len 10, Id 50%). A more stringent cutoff was defined when a pair had a BLAST alignment of length 50 with 80% identity or better (Len 50, Id 80%). These scores were chosen based on the authors’ previous experience with MADBlast, and multiple scores were chosen to avoid sensitivity to any one set of cutoffs. Because the BLAST algorithm masks low complexity sequences, many functions did not match even themselves. Failure to self-match was used as a functional filter to eliminate sequences that exhibit low complexity.

Binaries are composed of many functions, each treated as individual sequences. To aggregate the results of function-level similarity to a score of similarity of two binaries, denoted as A and B below, results are reported as a fraction where the denominator is the number of sequences for binary A that had *any* alignment above the chosen stringency level. The numerator for each similarity is the number of sequences in A that matched any function in B at the given stringency level. A perfect score of 1.0 indicates that all of the functions that had any alignment matched A to B. A poor score of 0.5 means that only ½ of the functions from A matched a sequence in B, and ½ matched other targets.

2) *Reimplementing Serial BLAST*

To ensure that biological assumptions were not being imposed on non-biological datasets, and to enhance reuse of the BLAST algorithmic core, the authors developed a BLAST implementation that is free of biological assumptions called MADBlast. Both the algorithmic redesign and the parallel driver that accompanies it are described below.

BLAST supports many different modes of running (*e.g.* DNA vs. DNA comparisons and protein vs. protein comparisons). But for the purposes of analyzing executable binaries, the only functionality required was the ability to compare protein sequences—the *blastp* operating mode. Several code features were essential for this refactored BLAST implementation:

- Memory is allocated for the dynamic programming calculations (key alignment algorithm) once up front and reused for each alignment.
- Large sequences are split, based on the amount of memory available and processed in pieces and reassembled as needed.
- Arbitrary alphabets up to about 80 characters are supported, using standard ASCII characters, with several restrictions caused by file format constraints.
- User-specified scoring matrices are selected at run-time, as well as a few other “tuning” variables that are now easily configurable and no longer hard-coded.

3) *Making Serial BLAST Parallel*

A key feature of MADBlast that enables the analysis described in this paper is parallelization using ZeroMQ [20]. This allows large corpuses of binaries to be analyzed quickly using a cluster. Note that this is only necessary when

Authorized licensed use limited to: IEEE Xplore. Downloaded on October 19,2024 at 16:56:54 UTC from IEEE Xplore. Restrictions apply.

performing the initial similarity score generation. During normal operation a single core is sufficient to compare unknown binaries to groups of known binaries.

D. Data

The binaries used in the example application of this binary similarity method come from the Pacific Northwest National Laboratory Molecular Science Computing Facility, a production computing HPC center that focuses on environmental and molecular science calculations. This center supports many users who develop and run a variety of computational chemistry and other codes. To simulate a whitelisting application, a sample of the executable binaries compiled for this system was obtained and analyzed to quantify the extent to which binaries known to be similar were found to be similar using the two methods presented. This collection of binaries had five different computational chemistry codes, each having a different number of variants, and one instance of a bioinformatics application. The largest family was the computational chemistry package, Amsterdam Density Functional (ADF) [21], having 22 different versions on the system. Because of this large number of versions, ADF was the family of interest in this study. Other codes were treated as "out groups" and included Amber [22], CP2K [23], VASP [24], Lammps [25], and ScalaBLAST [26, 27]. Table 1 contains a summary of code types and number of variants.

Table 1. Description of Software Binaries

Code Family	# Variants	Code Type
ADF	22	Chemistry
Amber	8	Chemistry
CP2K	3	Chemistry
VASP	2	Chemistry
Lammps	2	Chemistry
ScalaBLAST	1	Biology

After disassembly, there were 520,060 functional blocks across the code corpus represented as distinct biosequences. Table 2 contains the number of sequences that remained in this dataset after using various pre-BLAST length filters (# Seq.), the number that exhibited enough complexity to be aligned (# Cplx), and the numbers of sequences that had any alignment at either the less or more stringent alignment levels.

Table 2. Alignments from Sequences of Varying Lengths

Pre-BLAST cutoff	# Seq.	# Cplx	Len 10 Id 50%	Len 50 Id 80%
10	377907	129999	129060	68887
50	204557	112637	111976	69218
100	153110	99559	98955	64521

E. Assessing Statistical Performance

Using each binary as a classifier, the Receiver Operator Characteristic (ROC, which is a plot of true positive vs. false negative as similarity cutoff varies from 1 to 0) was calculated by sorting the similarity scores for that binary against all binaries in a test set. Area under the ROC curve (AUC) was used as a measure of statistical performance, with 1.0 being a perfect score. The similarity threshold required to correctly identify 90% of the true members of a family (T_{90}) was calculated by using the sorted similarity list for each binary and locating the similarity value at which at least 90% of true positives had been identified. Because ADF was the largest code family, having 22 different members, ADF was treated as the positive group, and all non-ADF instances were treated as the negative group.

Statistical significance of AUC and T_{90} results for feature vector and bio-based techniques were tested using one-way analysis of variance (ANOVA) to identify which families and subfamilies had statistical performance differences. For families or subfamilies determined by ANOVA to have statistically significant differences, Tukey's honest significant difference (HSD) method was used to identify which technique pairs produced the statistically significant differences. For both ANOVA and Tukey's HSD test, significance value of 0.05 was used.

III. Results

Applying hierarchical clustering to the MADBlast output revealed that there is a strong family similarity within three subgroups of ADF individuals. Figure 2 illustrates the results of this family tree with all ADF family members in grey or black, and all non-ADF individuals in white.

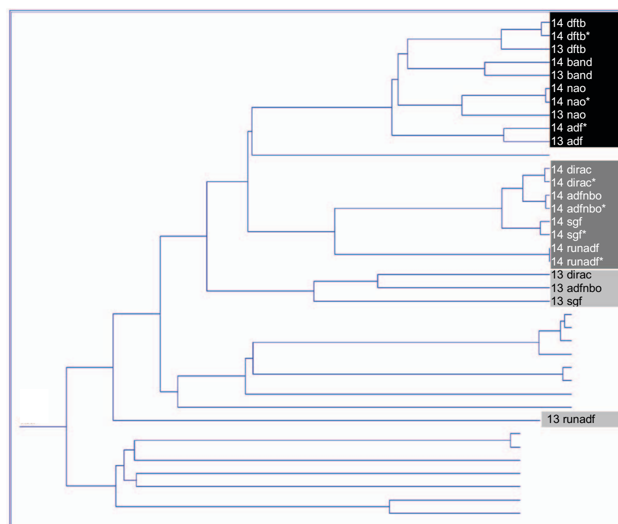

Figure 2. Dendrogram of code families by sequence similarity. Black and dark grey are strong subfamilies, each having only ADF instances (ADF family 2 and ADF family 3, respectively). Light grey are outlier ADF instances, and white are non-ADF codes**.**

Authorized licensed use limited to: IEEE Xplore. Downloaded on October 19,2024 at 16:56:54 UTC from IEEE Xplore. Restrictions apply.

The first ADF group is a collection of versions from the year 2013 identified by the software maintainers as “dirac”, “adfnbo”, “sgf”, and “runadf”. This group is the outlier ADF group, shown in light gray in Figure 2 and does not have a strong family substructure. The second group (ADF subfamily 2) has 10 members, each with one or two versions from 2014 and one version from 2013 (labeled as “14”, “14*”, or “13”) and codes identified as “dftb”, “band”, “nao”, and “adf”. ADF Family 2 is marked in Figure 2 with black. The third group, shown in dark gray in Figure 3 (ADF subfamily 3), contains only versions of ADF from 2014, each with two different instances of “dirac”, “adfnbo”, “sgf”, and “runadf”. Both bio-based (shown) and vector based (results not shown) family trees yielded identical ADF families 2 and 3, and the same ADF outliers, and all non-ADF codes are outside the ADF subfamily structure. All of the non-ADF codes grouped into correct smaller families with only 2 exceptions—Vasp 4 and 5 are very different according to this analysis, and CP2K 2.4 versions are similar to each other, but very different than CP2K 2.5.

AUC for ROC curves was produced by comparing each ADF member with respect to the full ADF family, and within the combination of subfamilies 2+3, and for subfamilies 2 and 3 alone. The AUC for these are illustrated in Figure 3.

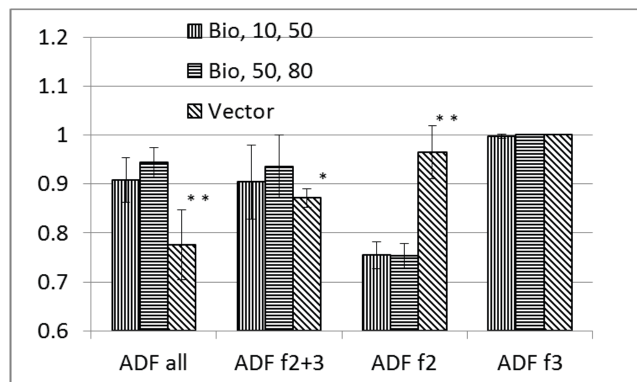

Figure 3. Mean AUC values for classifying all ADF instances (ADF all), ADF instances from two combined subfamilies (ADF f2+3) and individual subfamilies (ADF f2 and ADF f3) using the vector method, and the bioinformatics-based method with different filtering parameters. Statistically significant difference between vector and one bio-based method is indicated by *. Statistically significant difference between vector and both bio-based methods is indicated by **.

When all ADF members from disparate subfamilies are combined (ADF all results), the mean AUC for using any member of this family to identify all other members is above 0.9 using the bioinformatics-based method, regardless of whether stringent alignment (Len 50, Id 80) or less stringent alignment criteria (Len 10, Id 50) were used. By comparison, the feature vector based approach yielded a worse performance that is statistically significant when compared to either bio-based stringency classifier. To explore the effects of refining the family definition using the family tree results, the performance of both biosequence-based stringency levels and the feature vector approach were assessed on a combination of ADF family 2 and family 3. This is a more tuned family that does not contain the 4 ADF outliers. The results in Figure 3 for combining ADF families 2 and 3 (ADF f2+3) show that the stringent bio-based approach performed better than the feature vector based approach at a level achieving statistical significance. However, when we limited the analysis to only ADF family 2 (ADF f2), the feature vector approach statistically outperformed both the biosequence-based stringency levels. When the analysis was restricted to only ADF family 3 (ADF f3), all three methods performed nearly perfectly, and there was no statistically significant difference between them.

Another measure for family classifiers to identify members in a highly varied family is T_{90}. In general, a method that identifies most of its family members with a higher degree of similarity may fail to find new distant relatives of an established family. Figure 4 illustrates the relative T_{90} values for both biosequence-based stringency levels and the feature vector based approach.

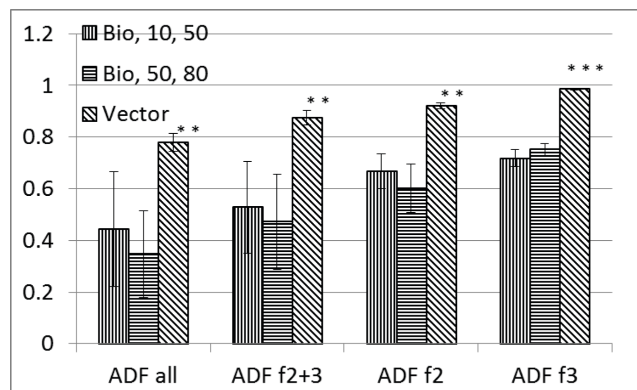

Figure 4. Mean T_{90} values for classifying all ADF instances (ADF all), ADF instances from two combined subfamilies (ADF f2+3) and individual subfamilies (ADF f2 and ADF f3) using the vector method, and the bioinformatics-based method with different filtering parameters. Statistically significant difference between vector and both bio-based methods is indicated by **. Statistically significant difference between all three methods is indicated by ***.

When identifying all ADF members together, (ADF all, where the biosequence methods both outperformed the feature-vector based method as assessed by AUC), the feature vector T_{90} was statistically significantly higher than that for both the biosequence-based methods. Reducing heterogeneity in the family by excluding the ADF outliers (ADF f2+3) reduced the difference between T_{90} for the feature vector approach and the biosequence-based approach, but the difference was still statistically significant. Considering the next more homogeneous families (ADF f2 and ADF f3), the difference inT_{90} between the feature vector and biosequence-based similarity measures decreased further, but in both cases still achieved statistical significance. The authors interpret these results to suggest that although the feature vector

Authorized licensed use limited to: IEEE Xplore. Downloaded on October 19,2024 at 16:56:54 UTC from IEEE Xplore. Restrictions apply.

method may sometimes outperform the biosequence-based method for highly tuned subfamilies of binaries, the biosequence-based approach in general is more flexible in identifying members in variant families.

IV. Discussion and future directions

In this paper, a pair of algorithm designs is presented for identifying similar binary executables, and these methods are demonstrated in an application of finding similar binaries in multi-user architectures such as HPC centers or cloud platforms. One method is based on feature vectors of disassembled instructions and the other is based on a biosequence-based approach. The feasibility of these methods was demonstrated by applying the methods to the challenge of identifying variants of a computational chemistry code in a collection of real-world software from an operational HPC system that included other chemistry binaries of the same family and from other families as well as a biology code.

This is a significant deviation from most clone detection methods because much of the syntactic and structural information is deliberately ignored from the original binaries. Instead, the clone detection problem is turned on its head to answer, "Are there highly conserved patterns between two binary streams that are more likely than one would expect by chance?" The answer to this question yields information about *commonalities* that may be meaningful within software families and signatures of these families. Sequence analysis for binaries lends itself directly to these types of analysis because they are well-studied problems for protein systems. This is where the mapping from binary analysis to protein sequence analysis provides value—family-based analysis techniques and motif-finding (a single representation of a family) capabilities are immediately applicable as a consequence of using mature bioinformatics approaches.

Analysis presented here shows that using the bioinformatics-based method to create classifiers using known instances of a binary can reliably identify many variants of the binary, even when those variants are built with different functionality. Using family tree analysis on the binary family gave insight into subgroups that were treated as subfamilies, and similar classification results were obtained when the subfamilies had members of varying composition.

The three most significant findings are 1) bioinformatics-based method statistically outperforms the feature-vector-based method when the family of code is larger and more variant; 2) the similarity measure from both the feature vector and biosequence-based approaches yield nearly identical family trees for an example binary corpus; and 3) using the family tree to refine the family definition can improve the performance of either the biosequence-based or feature vector-based identification method, in some cases to near-perfect statistical performance.

For the ADF example, the biosequence-based approach is statistically better in the general case when the subfamily structure is not known *a priori*. When this structure is known, it is possible to tune the performance of some subfamilies, potentially opening the door for a vector-based identification. However, because in the more general case of large, complex code families the biosequence-based approach is more flexible, for many other applications this would be the preferred method.

Using the biosequence-based similarity measure to analyze subfamilies, three interesting partitions were found in the collection of ADF binaries and one *orphan* variant of ADF existed in the binary collection. A set of code that does not fall into a family is considered an *orphan*. After talking with the maintainers of ADF on this system, the orphan was identified as an older driver code that was not surprisingly different than the others. The other differences in subfamily composition could be generally explained by differences in code version, or by functionality of the binary versions. The authors are in the process of investigating further into the possible meaning of the subfamilies identified via hierarchical clustering.

The statistical performance of the biosequence-based method on families and subfamilies is illustrative of the flexible but accurate ability of biological similarity algorithms to recognize familial variants. The presented application of identifying similar binaries in a production HPC environment is just one example of how the method could be used.

The authors plan to explore the use of the biosequence-based method on a variety of other applications including machine utilization measures during run-time. Though this would not be able to block inappropriate binaries from running, it should be able to detect inappropriate utilization of resources. It may be that users are running acceptable binaries in unacceptable ways, resulting in lower overall machine utilization or creating other resource bottlenecks. Characterizing the behavior of software with respect to hardware utilization may provide an alternative to the presented static binary analysis with an analogue of dynamic analysis.

The authors have shown in prior work that this method can be applied to applications that are most suited for blacklisting [28]. For example, a similar technique might be used for identifying binaries (or binary fragments) that should *not* be used on a system. In this case the signature is for a functional block, not for an entire binary. The bioinformatics approach is particularly well suited for this application because having models of disallowed binaries would make it possible to identify binaries being used that contain that disallowed functionality, even when it is embedded in a larger, seemingly acceptable application.

V. Conclusion

Presented herein is an example of a novel biosequence-based approach producing a reliable, flexible matching methodology for identifying similar executable binaries. This method was demonstrated on an example of a whitelisting application for verifying the identity of executable binaries using data from a live topical HPC system with a corpus of 6

Authorized licensed use limited to: IEEE Xplore. Downloaded on October 19,2024 at 16:56:54 UTC from IEEE Xplore. Restrictions apply.

different scientific codes, most having multiple versions on the system. Emphasis was placed on recognizing members of the largest family using other members of the family as an exemplar to assess the ability of this method to operate on highly complex, dynamic codes. For this application, the biosequence method statistically outperformed a simpler feature-vector based method for the binary family under study, and when two of the subfamilies were combined into a single classifier. One subfamily was well recognized by both the biosequence and feature vector methods, and for another subfamily the vector method outperformed the biosequence method. In all cases the threshold score required to detect 90% of the family members was higher (more strict) when using the feature vector based method, suggesting that regardless of false positive rate, the biosequence method was able to correctly find true positives using a more relaxed similarity threshold. Using the HPC implementation of this biosequence-based process, the structure of this family was rapidly learned to guide tuning of the identification process, resulting in a highly accurate and sensitive identification of software family members. Ultimately, this technique could be applied to a wide variety of applications in executable binary characterization and identification.

ACKNOWLEDGMENT

Thanks to Gary Skouson, Dave Cowley, Ping Yang, and Doug Baxter for providing sample executable binaries from the EMSL computing center and for helpful discussions on code differences.

REFERENCES

[1] Oh, J., *Fight against 1-day exploits: Diffing binaries vs anti-diffing binaries*, in *Blackhat technical security conference*. 2009.
[2] Kim, J.-B. and R.-H. Park, *Unsupervised binary hashing method using locality preservation and quantisation error minimisation.* Electronics Lett., 2015. **51**(3): p. 255-257.
[3] Prechelt, L., G. Malpohl, and M. Philippsen, *Finding plagiarisms among a set of programs with JPlag.* J. Universal Comput. Sci, 2002. **8**(11): p. 1016-1038.
[4] Li, Z., et al., *CP-miner: a tool for finding copy-paste and related bugs in operating system code.* IEEE trans. software eng., 2006. **32**: p. 176-192.
[5] Cohen, C. and J. Harvilla, *Function hashing for malicious code analysis*, in *CERT research report*. 2009, CERT Software Engineering Institute, Carnegie Mellon. p. 26-29.
[6] French, D., *Beyond Section Hashing*, in *CERT research report*. 2010, CERT Software Engineering Institute, Carnegie Mellon. p. 64-66.
[7] Harvilla, J., *Large-scale analysis of malicious PDF documents*, in *CERT research report*. 2010, CERT Software Engineering Institute, Carnegie Mellon. p. 67-68.
[8] Sæbjørnsen, A., et al., *Detecting code clones in binary executables*, in *ISSTA 2009*. 2009, ACM: Chicago, IL. p. 117-127.
[9] Casey, W., C. Cohen, and C. Hines, *Malware family analysis: correlating static features and dynamic characteristics on large-scale projects*, in *CERT research report*. 2010, CERT Software Engineering Institute, Carnegie Mellon. p. 61-63.
[10] Ding, W., et al., *KLONOS: Similarity-based planning tool support for porting scientific applications.* Concurrency and Computation: Practice and Experience, 2013. **25**: p. 1072-1088.
[11] Ding, W., et al., *Bio-inspired similarity-based planning support for the porting of scientific applications*, in *4th Workshop on Parallel Architectures and Bio-inspired Algorithms in conjunction with PACT11*. 2011.
[12] Needleman, S. and C. Wunsch, *A general method applicable to the search for similarities in the amino acid sequence of two proteins.* J. Mol. Biol., 1970. **48**(3): p. 443-453.
[13] Smith, T., M. Waterman, and C. Burks, *The statistical distribution of nucleic acid similarities.* Nucleic Acids Res., 1985. **13**(2): p. 645-656.
[14] Altschul, S., et al., *Basic local alignment search tool.* J. Mol. Biol., 1990. **215**(3): p. 403-410.
[15] Altschul, S., et al., *Gapped BLAST and PSI-BLAST: a new generation of protein database search programs.* Nucleic Acids Res., 1997. **25**(17): p. 3389-3402.
[16] Eagle, C., *The IDA Pro Book: The Unofficial Guide to the World's Most Popular Disassembler*. 2008, San Francisco, CA: No Starch Press.
[17] Panas, T. and D. Quinlan, *Techniques for software quality analysis of binaries: applied to Windows and Linux*, in *DEFECTS '09- 2nd international Workshop on Defects in Large Software Systems*. 2009. p. 6-10.
[18] Peterson, E., et al., *Novel Visual and Analytical Methods in Repurposing Legacy Scientific Code- A Case Study*, in *2013 International Conference on Software Engineering Research and Practice*. 2013.
[19] Bouckaert, R., et al., *WEKA-- Experiences with a Java Open-Source Project.* J. Machine Learning Research, 2010. **11**: p. 2533-2541.
[20] Hintjens, P., *ZeroMQ: Messaging for Many Applications*. 2013: O'Reilly Media, Inc.
[21] te Velde, G., et al., *Chemistry with ADF.* J. Computational Chemistry, 2001. **22**(9): p. 931-967.
[22] Pearlman, D., et al., *AMBER, a package of computer programs for applying molecular mechanics, normal mode analysis, molecular dynamics, and free energy calculations so simulate the structural and energetic progerties of molecules.* Computer Physics Communications, 1995. **91**(1-3): p. 1-41.
[23] Hutter, J., et al., *CP2K: Atomistic Simulations of Condensed Matter Systems.* Wiley Interdisciplinary Reviews: Computational Molecular Science, 2014. **4**(1): p. 15-25.
[24] Hafner, J., *Ab-initio simulations of materials using VASP: Density-functional theory and beyond.* J. Computational Chemistry, 2008. **29**(13): p. 2044-2078.
[25] *http://lammps.sandia.gov*.
[26] Oehmen, C. and J. Nieplocha, *ScalaBLAST: A Scalable Implementation of BLAST for High-Performance Data-Intensive Bioinformatics Analysis.* IEEE trans. Parallel and Distributed Systems, 2006. **17**(8): p. 740-749.
[27] Oehmen, C. and D. Baxter, *ScalaBLAST 2.0: Rapid and Robust BLAST Calculations on Multiprocessor Systems.* Bioinformatics, 2013. **29**(6): p. 797-798.
[28] Oehmen, C., E. Peterson, and J. Teuton, *Evolutionary drift models for moving target defense*, in *Eighth Annual Workshop on Cybersecurity and Information Intelligence Research*. 2012: Oak Ridge, TN.

Authorized licensed use limited to: IEEE Xplore. Downloaded on October 19,2024 at 16:56:54 UTC from IEEE Xplore. Restrictions apply.