PHYLOGENETIC ANALYSIS OF SOFTWARE USING CACHE MISS STATISTICS

Sebastiano Verde, Simone Milani, Giancarlo Calvagno

University of Padova, Department of Information Engineering

ABSTRACT

While the phylogenetic analysis of multimedia documents keeps being investigated, some recent studies have shown the possibility of re-using the same strategies to analyze the evolution of computer programs (Software Phylogeny), considering its many applications spanning from copyright enforcement to malware detection.

This paper presents a solution for reconstructing the phylogenetic dependencies of different releases of a given program. The proposed method collects cache miss statistics during the program execution, builds a dissimilarity matrix from the results, and then estimates the corresponding Software Phylogenetic Tree (SPT) using a minimum spanning tree algorithm.

***Index Terms*—** phylogeny, cache management, software forensics, software identification, multimedia forensics

1. INTRODUCTION

The online diffusion of documents and multimedia contents reveals several analogies with the genetic evolution of living organisms. As an example, a posted image can be downloaded, modified and re-posted online several times. In these operations, users may process either the original picture or one of its other versions. This generates a set of near-duplicate copies that can be well represented by the nodes of a tree (called Phylogenetic Tree or PT), where edges are associated to transformations. Since this tree structure could not be completely recovered from the metadata available on the network, recent studies in the information forensics literature have tried to reconstruct these chains of processing steps using *phylogenetic analysis* strategies. Given a set of near-duplicate multimedia contents (like images [1], videos [2] and audio files [3]), it is possible to compute a set of similarity/dissimilarity values on each couple of similar contents and recover the original PT structure by means of a maximum/minimum spanning tree algorithm.

More recently, some preliminary results have shown that these methodologies can be applied to software as well [4]. In fact, computer programs usually evolves progressively, including additional functionalities or reducing the computational load thanks to a rewriting of the most frequently-used pieces of code. This can be easily verified on programs and libraries developed using web-based project hosting services [5]; several programmers collaborate on the final code by means of small incremental changes. Moreover, several parallel versions are sometimes developed carrying on different branches together with the main implementation. A phylogenetic structure can also be verified in the evolution of malwares [6, 7], which usually change operation orders and function calls with the purpose of mismatching the signatures stored in the scanning programs. Reconstructing the software phylogenetic tree (SPT) allows verifying their origins, and therefore identifying the most appropriate measures to make them harmless. In the end, localizing a piece of software within its phylogenetic tree contributes to the parametrization of its originality and to evaluate possible patent violations.

This paper presents a system for the reconstruction of the SPT relative to different types of programs. The similarity between different versions are measured with respect to the statistics of cache misses measured in different working conditions. This allows to extend such analysis to heterogeneous software, differently from the preliminary phylogenetic approach in [4] that only considered multimedia codecs, through an analysis of the processed signals. Moreover, the reconstruction accuracy was improved by fusing multiple phylogenetic trees originated from different inputs.

In the following, Section 2 overviews some related works, while Section 3 described the adopted algorithm. Experimental results are reported in Section 4, and final conclusions are drawn in Section 5.

2. RELATED WORKS

The phylogenetic analysis of online contents was initially focused on the investigation of near-duplicate documents [8], and later extended to images [1], videos [9, 10, 2] and audio files [3, 11]. In these analysis, the definition of a reliable metric to parameterize the similarity/dissimilarity between couples of contents is crucial. As for images, near-duplicate contents usually differ by means of an affine transformation [12], which allows modelling the dissimilarity with a distance function after image registration. Since these measures result quite noisy, additional processing is required in order to make the estimation more robust [13, 14]. When dealing with

This work has been partially supported by the University of Padova project Phylo4n6 prot. BIRD165882/16.

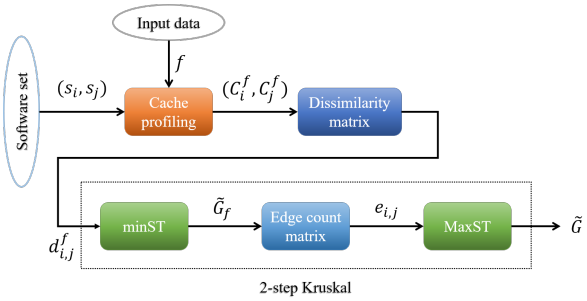

Fig. 1. Proposed software phylogeny pipeline.

video contents, alignment and dissimilarity computation are performed by means of local descriptors [15], while the work in [16] employs a set of transform based features that targets compressed audio files.

Some recent works focus on the problem of reconstructing the phylogenetic tree for multimedia codecs [4]; dissimilarity measurements were computed on the outcomes of coding operations. Although effective, such strategy could not be extended to other kind of software solutions. As a matter of fact, we had to focus on other types of distinctive features to measure software similarities.

Some previous works on malicious software detection and identification computed a set of characterizing features from the sequence of coded instructions [17]. In order to overcome the problem of obfuscation (which is largely employed for malware and viruses), it is possible to adopt a dynamic feature synthesis [18], while others process software outcomes like register errors [19].

These solutions proved to be extremely robust to obfuscation, re-engineering and partial alterations, which make the authentication and plagiarism detection extremely difficult as the complexity of the software increases.

3. PHYLOGENETIC ANALYSIS

Given a set $\mathcal{S} = \{s_1, \ldots, s_N\}$ of N different pieces of software, the phylogenetic analysis of $\mathcal{S}$ is usually divided in two main steps.

First, we need to define a suitable dissimilarity metric to measure the amount of differences between a pair of programs. By computing the dissimilarity $d_{i,j}$ for each pair (s_i, s_j), $\forall i \neq j$, it is possible to arrange the resulting values in a $N \times N$ dissimilarity matrix $D = [d_{i,j}]$. Secondly, the obtained dissimilarity matrix D is processed by a minimum spanning tree (MST) algorithm, which returns the most likely phylogenetic graph for the given software set $\mathcal{S}$.

Typical MST algorithms for weighted undirected graphs are Kruskal's [20] and Prim's [21]. In this work, we propose an original solution based on a twofold application of Kruskal's algorithm, as described in 3.2.

3.1. Dissimilarity metric

The first stage of a phylogenetic analysis consists in measuring the dissimilarity between pairs of objects by means of a properly defined metric. For analyzing software, we propose a dissimilarity metric that leverages cache miss statistics collected during the execution of the program.

A cache miss is a failed attempt to read or write data in the CPU cache memory, which results in a main memory access and a longer latency. Cache miss statistics provide useful information to perform memory debugging and software profiling, and can be collected through the use of specific tools during the execution of a piece of software. For this work we employed Cachegrind, one of the tools available in the framework Valgrind [22].

Cachegrind analyzes how the program interacts with a cache memory by simulating a machine with independent first-level instruction and data caches (I1 and D1), backed by a unified second-level cache (L2), which matches the configuration of many modern machines. For machines having more than two levels of cache, Cachegrind simulates the first-level (L1) and last-level (LL) caches only, since the LL cache has the most influence on runtime, as it masks accesses to main memory, and the L1 caches often have low associativity, so simulating them can detect bad code interactions.

Statistics collected by Cachegrind are reported in Table 1.

I_r	Number of instructions executed
$I1_{mr}$	I1 cache read misses
IL_{mr}	LL cache instructions read misses
D_r	Number of memory reads
$D1_{mr}$	D1 cache read misses
DL_{mr}	LL cache data read misses
D_w	Number of memory writes
$D1_{mw}$	D1 cache write misses
DL_{mw}	LL cache data write misses

Table 1. Cachegrind statistics.

These statistics are presented for each function in the program and therefore it is possible to arrange the output of the simulation in a $M \times 9$ *cache statistics matrix* C, where M equals the number of functions in the program.

Operationally, our system takes a program $s_i \in \mathcal{S}$ and an input file f from a certain set $\mathcal{F}$ (for programs that requires some input) and runs the Cachegrind simulation, providing a cache statistics matrix C_i^f. This is repeated for each program in $\mathcal{S}$.

Given a pair of cache statistics matrices (C_i^f, C_j^f), we define the dissimilarity between them as

$$d_{i,j}^f = \|C_i^f - C_j^f\|_2, \tag{1}$$

where $\|\cdot\|_2$ denotes the matrix 2-norm. Note that different programs (and even different releases of the same software)

use different functions, in general, which means that cache statistics matrices may have incompatible dimensions. To overcome this problem, we tested two approaches: i) taking the *intersection* of the two function sets; ii) taking the *union* of the two function sets. In the latter case, functions missing in one program are filled with a zero-vector in the cache statistics matrix.

Since a program may behave differently in response to the specific input, we collect a set of dissimilarity matrices,

$$D^f = \left[d^f_{i,j} \right],$$

for multiple input files $f \in \mathcal{F}$ in order to achieve a higher accuracy. These matrices are then processed by the phylogenetic tree reconstruction algorithm, described in the following paragraph.

3.2. Phylogenetic tree estimation

In the second stage of a phylogenetic analysis, the dissimilarity metrics computed in the previous block are usually processed by a MST algorithm that returns an estimated phylogenetic tree. However, our system produces multiple dissimilarity matrices, one for each input file, and therefore a proper fusion technique has to be designed.

We propose a procedure called *2-step Kruskal* (2K), built on the classic Kruskal's algorithm, which takes in input a set of weighted adjacency matrices an outputs a single MST. The Kruskal's algorithm is widely adopted in phylogenetic reconstruction, although other solutions can be employed and adapted to our double MST approach.

The 2K algorithm consists of three stages:

1. $\forall f \in \mathcal{F}$, run Kruskal's algorithm on D^f, obtaining the minimum spanning tree $\hat{G}^f$;

2. $\forall (i,j)$, count how many times the edge i,j is chosen among all the estimated trees $\hat{G}^f$ and store the results in a matrix $E = [e_{i,j}]$;

3. run Kruskal's algorithm on $-E$, obtaining the maximum spanning tree of E, corresponding to the final phylogenetic tree, $\hat{G}$.

This procedure practically consists in a majority voting of the most frequently reconstructed tree. Instead of simply averaging the matrices D^f and computing a single MST, this method proved to offer better stability against the presence of a few noisy matrices, as shown in Section 4. Furthermore, matrix averaging presents non-trivial normalization issues since different dissimilarities have different magnitudes. Thanks to the designed fusion strategy, the 2K approach overcomes this problem.

Thor com.	**OpenJPEG rel.**	**RNNoise com.**
2015/11/17	2.0	2017/08/20
2016/01/18	2.1	2017/09/08
2016/03/11	2.1.1	2017/09/15
2016/03/21	2.1.2	2017/09/20
2016/05/28	2.2.0	2017/10/18
2016/11/09	2.3.0	
2017/01/20		
2017/03/22		
2017/10/21		
2018/01/17		
LZ4 rel.	**LIBSVM rel.**	
1.7.3	v270	v310
1.7.4.2	v287	v311
1.7.5	v300	·
1.8.1	v311	·
1.8.1.2	v316	·
1.8.2	v317	·
1.8.3	v323	v323

Table 2. Releases and commits of analyzed software.

4. EXPERIMENTAL RESULTS

The validation of the proposed SPT estimation strategy was carried out by analyzing the releases and commits of a set of open source programs.

We considered two multimedia codecs already studied in [4], namely Thor [23] and OpenJPEG [24], as well as other kinds of programs: LZ4 [25], a lossless data compressor; RNNoise [26], a recurrent neural network for audio noise reduction; and LIBSVM [27], a library for support vector machines. The complete list of commits and releases is reported in Table 2. Note that for LIBSVM we considered two sets of releases, one with major releases only, and one with all the minor releases from v310 to v323.

Furthermore, we built different sets of input files to meet the requirements of each program. For Thor, we used 7 raw video sequences at 4CIF resolution (`crew`, `crowdrun`, `duckstakeoff`, `harbour`, `ice`, `parkjoy`, `soccer`). For OpenJPEG and LZ4, 30 images from the Kodak dataset [28] were employed. LIBSVM was tested with 5 datasets for binary classification.

The evaluation of the estimated SPTs was performed by means of the percentage of correctly reconstructed edges. Each program was tested with four different strategy configurations: *union* or *intersection* of function sets, for dissimilarity computation; *2-step Kruskal* algorithm or *dissimilarity averaging*, for tree estimation.

The results for every program and strategy configuration are reported in Table 3, where $2K$ denotes the proposed 2-step Kruskal strategy and $\bar{D}$ denotes dissimilarity averaging. Moreover, the comparison with the pixel-based strategy proposed in [4] is reported for Thor and OpenJPEG. Note how

Analyzed software				Percentage of correct edges				
Name	Type	No. releases	No. inputs	$2K_{\text{union}}$	$\bar{D}_{\text{union}}$	$2K_{\text{inters}}$	$\bar{D}_{\text{inters}}$	[4]
Thor	Video coding	10	7	0.89	0.89	0.67	0.78	0.56
OpenJPEG	Image coding	6	30	0.60	0.60	0.60	0.60	0.20
RNNoise	Audio denoising	5	40	0.50	0.25	0.50	0.25	-
LZ4	Data compression	7	30	0.67	0.67	0.67	0.67	-
LIBSVM	Machine learning	7	5	0.83	0.83	0.50	0.50	-
		14	11	0.23	0.15	0.23	0.15	-

Table 3. Experimental results for analyzed software.

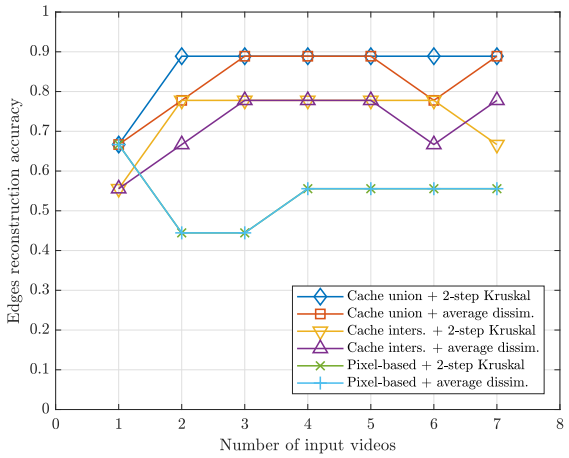

Fig. 2. Accuracy comparison of different dissimilarity metrics and tree reconstruction strategies for Thor software (10 releases) with increasing number of inputs.

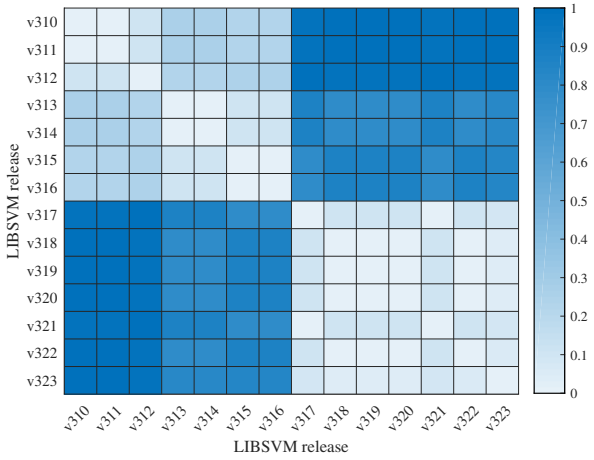

Fig. 3. Heatmap visualization of the average dissimilarity matrix for LIBSVM software (releases from v310 to v323), showing an evident clustering into two subsets of similar releases.

2K algorithm combined with the union of cache statistics matrices performs best (or on par) for all the analyzed programs.

Figure 2 reports the accuracy for the analysis of Thor video codec releases as we increase the number of input files. It is possible to see how the cache union with the 2K solution not only outperforms (almost) any other configuration, but also provides the most stable reconstruction, yielding a constant accuracy for each number of input videos greater than 2.

Finally, Figure 3 shows a heatmap representation of the average dissimilarity matrix obtained for the second set of LIBSVM releases (containing all minor releases), from which we can observe an additional property of the proposed system. This set is particularly tricky for phylogeny since the releases are extremely similar to one another. However, two major releases, v316 and v317, are contained in the set, and the clusterization between pre-v316 and post-v317 releases is clearly visible from the dissimilarity matrix. In other words, the system is able to distinguish whether a significant implementation change happened or a minor commit or bug fix since the dissimilarity metric scales in accordance with the amount of change existing between two pieces of software.

5. CONCLUSIONS

In this work we presented a solution to the software phylogeny problem, which employs cache miss statistics to measure the dissimilarity between pairs of programs, and estimates the phylogenetic structure by means of a minimum spanning tree algorithm. This method improves the state of the art, as it allows to deal with generic pieces of software, other than multimedia codecs. The proposed system was validated with releases and commits of various open source programs available online. Experimental results show a promising accuracy in the phylogenetic reconstruction, as well as in distinguishing major releases from minor commits and bug fixes. Future work will be devoted to evaluating how the reconstruction accuracy varies for different operating systems and compilers and to parameterizing software dissimilarity metric including register values and errors to enhance the reconstruction accuracy.

6. REFERENCES

[1] Z. Dias, A. Rocha, and S. Goldenstein, "First steps toward image phylogeny," in *Proc. of 2010 IEEE WIFS*, Dec. 2010, pp. 1 –6.

[2] S. Milani, P. Bestagini, and S. Tubaro, "Video phylogeny tree reconstruction using aging measures," in *Proc. of EUSIPCO 2017*, 2017, pp. 2181–2185.

[3] S. Verde, S. Milani, P. Bestagini, and S. Tubaro, "Audio phylogenetic analysis using geometric transforms," in *Proc. of IEEE WIFS 2017*, Dec 2017, pp. 1–6.

[4] Sebastiano Verde, Simone Milani, and Giancarlo Calvagno, "Phylogenetic analysis of multimedia codec software," in *Proc. of EUSIPCO 2018*, Sept. 2018.

[5] L. Dabbish, C. Stuart, J. Tsay, and J. Herbsleb, "Social coding in github: Transparency and collaboration in an open software repository," in *Proc. of 2012 ACM CSCW*, 2012, pp. 1277–1286.

[6] Md. Enamul Karim, Andrew Walenstein, and Arun Lakhotiaand Laxmi Parida, "Malware phylogeny generation using permutations of code," *J. Comput. Virol.*, vol. 1, no. 1-2, pp. 13–23, Nov. 2005.

[7] D. Iliopoulos, C. Adami, and P. Szor, "Darwin inside the machines: Malware evolution and the consequences for computer security," *CoRR*, vol. arXiv 1111.2503, 2011.

[8] L. Kennedy and S.-F. Chang, "Internet image archeology," in *Proc. of ACMMM*, 2008.

[9] John R. Kender, Matthew L. Hill, Apostol Natsev, John R. Smith, and Lexing Xie, "Video genetics," in *Proc. of ACMMM*, 2010.

[10] F. Costa, S. Lameri, P. Bestagini, Z. Dias, S. Tubaro, and A. Rocha, "Hash-based frame selection for video phylogeny," in *Proc. of 2016 IEEE WIFS*, Dec. 2016, pp. 1–6.

[11] S. Verde, N. Pretto, S. Milani, and S. Canazza, "Stay true to the sound of history: Philology, phylogenetics and information engineering in musicology," *Applied Sciences*, vol. 8, no. 2, 2018.

[12] A. De Rosa, F. Uccheddu, A. Costanzo, A. Piva, and M. Barni, "Exploring image dependencies: A new challenge in image forensics," in *Proc. of SPIE*, 2010.

[13] A. Melloni, P. Bestagini, S. Milani, M. Tagliasacchi, A. Rocha, and S. Tubaro, "Image phylogeny through dissimilarity metrics fusion," in *Proc. of EUVIP 2014*, 2014, pp. 1–6.

[14] S. Milani, M. Fontana, P. Bestagini, and S. Tubaro, "Phylogenetic analysis of near-duplicate images using processing age metrics," in *Proc. of 2016 IEEE ICASSP*, 2016, pp. 2054–2058.

[15] S. Lameri, P. Bestagini, A. Mellon, S. Milani, A. Rocha, M. Tagliasacchi, and S. Tubaro, "Who is my parent? reconstructing video sequences from partially matching shots," in *Proc. of 2014 IEEE ICIP*, 2014, pp. 5342–5346.

[16] M. Maksimovic, L. Cuccovillo, and P. Aichroth, "Phylogeny analysis for mp3 and aac coding transformations," in *Proc. of 2017 IEEE ICME*, 2017, pp. 1165–1170.

[17] A. Palisse, A. Durand, and J.-L. Lanet, "Malware'O'Matic a platform to analyze Malware," in *French Japanese workshop on CyberSecurity*, Rennes, France, Sept. 2016, Inria.

[18] F. Biondi, S. Josse, and A. Legay, "Bypassing malware obfuscation with dynamic synthesis," *ERCIM News*, , no. 106, Sept. 2016.

[19] G. E. Dahl, J. W. Stokes, L. Deng, and D. Yu, "Large-scale malware classification using random projections and neural networks," in *Proc. of 2013 IEEE ICASSP*, May 2013, pp. 3422–3426.

[20] Joseph B. Kruskal, "On the shortest spanning subtree of a graph and the traveling salesman problem," *Proceedings of the American Mathematical Society*, vol. 7, no. 1, pp. 48–50, 1956.

[21] Robert C. Prim, "Shortest connection networks and some generalizations," *The Bell System Technical Journal*, vol. 36, no. 6, pp. 1389–1401, Nov. 1957.

[22] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," *SIGPLAN Not.*, vol. 42, no. 6, pp. 89–100, June 2007.

[23] Cisco System, "Thor Video Codec Git Hub Repository," `https://github.com/cisco/thor`, Dec. 2017.

[24] ISPGroup (UCL), "Open JPEG: An open-source JPEG 2000 codec," `https://github.com/uclouvain/openjpeg/`, 2017.

[25] LZ4, "Extremely Fast Compression Algorithm," `https://github.com/cisco/lz4/lz4`, 2018.

[26] Xiph.org Foundation, "RNNoise," `https://github.com/xiph/rnnoise`, 2018.

[27] Cisco System, "Thor Video Codec Git Hub Repository," `https://github.com/cisco/thor`, 2017.

[28] Kodak, "Kodak Lossless True Color Image Suite," `http://r0k.us/graphics/kodak`, 2017.