A comparative analysis about similarity search strategies for digital forensics investigations

Vitor Hugo Galhardo Moia and Marco Aurélio A. Henriques

Abstract— **Known File Filtering method separates relevant from non-relevant information in forensics investigations using white or black lists. Due to limitations on hash functions (inability to detect similar data), approximate matching tools have gained focus recently. However, comparing two sets of approximate matching digests using brute force can be too time-consuming. Strategies to efficiently perform lookups in digests databases have been proposed as a form of similarity search. In this paper, we compare some strategies based on ssdeep and sdhash tools concerning to precision, memory requirement, and lookup complexity. We show that none of these strategies address these requirements satisfactorily.**

Keywords— **Digital forensics, Approximate matching, similarity hash, similarity search, sdhash, ssdeep.**

I. Introduction

With the popularity of technology and increase in storage capacities, more and more data has been generated from a variety of sources: desktop computers, smartphones, tablets, among other devices. The result is an enormous amount of data dealt by forensics examiners even in ordinary investigations. Known File Filtering method comes up as one of the solutions to cope with this overwhelming volume of data. By using white/black lists of reference data, forensics can reduce and separate relevant from non-relevant information, usually with the help of hash functions (MD5, SHA-1, SHA-2 etc). NIST [1] supports this kind of procedure by providing databases of hashes of known objects, mainly related to white lists. However, hashes are fragile in the sense that small changes in an object will produce an entirely different digest. This way, one can insert a single byte in a malicious object to compromise the identification of this data in investigations. Besides, as objects are constantly changing and getting updates, such as operating system files, keeping a hash for every single change of an object is infeasible, due to the huge database size required, and the difficulties to keep it up to date.

Approximate matching functions are suitable candidates to mitigate these problems. They can identify similarity between objects in such a way that similar data will produce similar digests. Even if changes occur in data, the corresponding digest will change accordingly, making them an attractive solution for a vast sort of applications in digital forensics.

Despite its benefits, approximate matching functions are far more expensive than cryptographic hashes. They usually require two functions, one for generating digests and another to compare them. Also, some methods produce digests with length proportional to the input object size, increasing the storage requirements. However, the major bottleneck regarding these functions is not the generation process itself, but the search for similarity. The brute force method is the standard strategy for similarity search. Every object in the media under investigation is hashed using an approximate matching tool and the digest compared to every digest in the reference database, resulting in a very time-consuming process. To deal with this particular limitation, some similarity search strategies have been proposed in the literature.

In this paper, we compare some similarity search strategies based on the two most popular approximate matching tools: ssdeep and sdhash. Our comparison takes into consideration three requirements: precision, memory, and time complexity. In the end, we discuss our findings and show that a better matching strategy is needed as these requirements are not met satisfactorily by the approaches compared.

II. Approximate matching tools

NIST [2] defines approximate matching functions as a "Promising technology designed to identify similarities between two digital artifacts. It is used to find objects that resemble each other or find objects that are contained in another object". In this work, we restrict our research to only those functions operating at the byte level, since they are more efficient and format independent. The matching process relies only on the object bytes, and it does not try to interpret the object neither it considers any data structure.

In contrast to hashes which only give binary answers (the objects are identical or not), approximate matching functions provide a confidence measure about the similarity of two objects. Also, while hash functions produce small fixed-size digests, these functions usually create larger digests, not necessarily of fixed-size. Some tools generate digests with size proportional to the input data.

There is a vast number of applications for approximate matching schemes. They can be used to detect new versions of documents, libraries or software, embedded objects (e.g., image inside a document), code reuse, malware clustering, etc.

In the next subsections, we introduce the two approximate matching tools analyzed in this work: ssdeep and sdhash.

A. The ssdeep tool

CTPH (Content Triggered Piecewise Hashing) is a method to detect similarity between two objects. It was proposed

Vitor Hugo Galhardo Moia and Marco Aurélio A. Henriques, School of Electrical and Computer Engineering (FEEC), University of Campinas (UNICAMP), Campinas-SP, Brazil, E-mails: vhgmoia@dca.fee.unicamp.br, marco@dca.fee.unicamp.br. This work was partially supported by CAPES FORTE Project (23038.007604/2014-69).

by Tridgell and is based on the rsync [3] and spamsum [4] algorithms. In 2006, Kornblum implemented ssdeep [5], based on Tridgell's spamsum. The main idea of ssdeep is to break the input object into variable-size blocks and use their hash to represent the object. To this end, a rolling hash algorithm is used to set boundaries on the object (where blocks start and stop). A sliding window of fixed-size moves byte-by-byte in the input and, for each step, the rolling hash is used within the window content. Whenever this function produces a particular output, a trigger point is set, denoting the beginning and ending of blocks. All resulting blocks are hashed using an FNV algorithm [6], and the six least significant bits of each hash are encoded using base64 characters. The concatenation of all characters produces a final digest with at most 96 bytes (no metadata included).

The comparison function of ssdeep uses the Edit Distance algorithm, counting the minimum number of operations to transform one digest into the other, by weighted operations (insertion, deletion, substitution, and transposition). The result is scaled in the range of 0 to 100, where 0 (zero) means dissimilarity and 100, a perfect match.

The main disadvantage of ssdeep is that it only works for relatively small objects and with similar sizes. However, research has been conducted to address this and other limitations, as performance [7], [8] and security [9].

B. The Similarity Digest hash tool (sdhash)

Proposed by Roussev [10] in 2011, sdhash is another bytewise approximate matching tool to measure similarity. The main idea is to identify features (byte sequences) from an object that are least likely to occur in any other one by chance and use them to represent the object.

Roughly speaking, sdhash works as follows: The Shannon entropy is calculated for every feature (B-byte sequence) of the object. Then, a sliding window of W consecutive features scores points to those with the lowest entropy within the window [11]. All features with a score higher than a defined threshold are selected. There is a filtering process to eliminate weak features (the ones ranging in undesirable intervals) and to reduce false positives. Next, each remaining feature is hashed (SHA-1) and the result split into five sub-hashes. The 11 least significant bits of each sub-hash are used to address the bits within a Bloom filter. Each filter allows the insertion of a maximum number of elements due to false positive restrictions, and when it reaches its capacity, a new one is created. The final digest is a sequence of all Bloom filters. The digest length is proportional to the object size ($\approx 2.6\%$).

Comparing two sdhash digests means comparing their bloom filters. The first filter from the first digest is compared to every filter from the second one, selecting the maximum score. The process repeats for all remaining filters of the first digest. The result is an average from all comparisons, ranging from 0 (non-match) to 100 (very similar objects).

Recent research exposed vulnerabilities and presented improvements to sdhash [12], [13], [14]. However, its main disadvantage is the digests length and the time to generate them.

III. Strategies for Similarity Search

The major bottleneck in digital forensics investigations using the Known File Filtering method and approximate matching tools can be the similarity search process. In this approach, two datasets need to be compared to find similar items. The examiner holds a reference database and checks if any of the objects of this set is present in the analyzed media.

Unlike finding exact matches, which can be solved efficiently using ordinary databases, finding objects that resemble each other or are contained one in another is a much harder task. Comparing digests for detecting similarity requires a particular comparison function usually peculiar to the chosen approximate matching tool. Sometimes, the only option available is the brute force approach, a simple method where every object from the first set is compared to every object of the second one (all-against-all). The comparison is done by using the comparison function of the approximate matching tool and selecting the best matches above a certain threshold.

Although brute force works and can be used for any similarity tool, it is very time-consuming, becoming infeasible as the datasets sizes increase. In order to improve this process, some researchers came up with new strategies to perform the similarity search, which will be described next.

A. F2S2 strategy

Winter, C. et al. [15] proposed Fast Forensic Similarity Search (F2S2) as a similarity search strategy intended to be used with ssdeep, although not restricted to it. Using n-grams extracted from the digests, F2S2 builds an index structure and upon a query request, selects those digests sharing the same n-grams of the queried item. The comparison using the ssdeep tool is restricted to these selected candidates only. The authors related an impressive speedup compared to the brute force approach, above 2000 times faster.

The structure chosen to store the n-grams was a particular hash table, composed of a central array and variable size buckets that can store multiple entries. Each n-gram is a sequence of n consecutive bytes of the digest. The possible n-grams of a digest b of l bytes are: $b_1...b_n, b_2...b_{n+1}, ..., b_{l-n+1}...b_l$; they are used as lookup keys and provide a link to the digest sharing the same n-grams, and are made of two parts: e-key and bucket address. E-keys identifies the n-gram, while the bucket address is the position of the n-gram in the index table, got from a mapping function using the k leading bits of the n-gram (since ssdeep digests are base64 encoded, it is necessary to decode them before selecting the bits). The e-keys are stored in the buckets along with an ID, responsible for linking the corresponding digest with its n-grams [15].

Before performing any search using the F2S2 strategy, it is necessary to construct the index and insert all reference list objects. The digests are created for the objects using ssdeep and for each one, an ID is assigned. Next, n-grams are extracted from the digests and inserted in the index along with their IDs. When two n-grams of different digests point to the same bucket, a new entry in this bucket is created, and the n-gram is stored with both IDs, one followed by the other. When performing a similarity search, the n-grams of the queried item are extracted and searched in the index for similar ones. In the case of a match, the digests belonging to the IDs of the n-grams found are selected and later compared to the queried item using ssdeep to confirm their similarity.

B. The Bloom filter approach: MRSH-NET

Another similarity search strategy is proposed by Breitinger, F. et al. [16], called MRSH-NET. This approach works with sdhash but also supports other approximate matching tools [17]. By using a single, huge Bloom filter, MRSH-NET can represent all reference list objects. However, the limitation of this approach is that it only answers binary questions: Is file A contained in this set? If so, an affirmative answer is returned, but it does not point out what are the similar objects.

sdhash extracts features from one object and insert them in a sequence of Bloom filters. Using MRSH-NET strategy, the features extracted this way from all reference list objects are inserted into a single, huge Bloom filter. To decide whether an object is present in the set or not, the match decision takes into account a sufficiently large number of subsequent features that need to be found in the filter. Performing a search means extracting the features from an object and executing a membership query for each one of them. If more than a pre-defined number of consecutive features (threshold) is found in the filter, the object is said to belong to the reference set.

C. Bloom filter-based tree strategy

To mitigate the limitation of MRSH-NET in answering only membership queries, Breitinger, F. et al. [18] proposed a new approach for performing similarity search. This strategy is based on the well-known divide and conquer paradigm. They build a Bloom filter-based tree data structure to store digests from a reference list. However, this method exists only in theory and lacks a working prototype.

This new strategy aims at returning the actual matching objects. First of all, a set S with n elements is inserted in a single Bloom filter, which represents the root node of a tree. The elements, in this case, are the objects, which have their features extracted using sdhash. Then, the set s is recursively divided into x subsets with n/x items and each subset inserted in a new Bloom filter, a child of the previous node. In the end, we need to create the FI (File Identifier) in the leaves of the tree, which are links to a database containing the digests. FICs (File Identifier Counter) are also created along with the FIs, initially set to 0 (zero) and incremented in a lookup procedure when the corresponding FI is traced.

The Bloom filter-based tree data structure creation is simple: extract the features from every object of the reference list and then insert them into the Bloom filter tree, along with their corresponding metadata (FI and FIC).

With this strategy, a lookup procedure becomes fast since we only compare digests with a subset of nodes. We first extract the features from the queried object and look up for them in the root node. If we get a non-match, we know for sure that this feature is not in the structure and can proceed to the next one. However, if we get a match on the root node, we need to trace a path down to the tree ending at a leaf. We then increase the FIC of that particular feature. After checking all features from the object, the highest FIC value is compared to a threshold (minimum number of consecutive features found in the filter). If FIC is above the threshold, we can say that the object is present in the set and then take the corresponding FI to reach the related digest. A comparison of the queried item digest and the returned one is necessary to verify similarity, using the comparison function of the approximate matching tool. As a matter of fact, most comparisons will yield a non-match when dealing with blacklisting cases. The search, in such case, will end in the root node since the number of bad or illegal objects is usually much smaller than the total number of objects, making this strategy very time efficient.

IV. Comparison of the strategies

In this section, we present our comparisons among the similarity search strategies developed for ssdeep and sdhash. We additionally introduced the brute force method using both tools. We aim at comparing three main aspects: Tool's precision, memory requirements, and lookup complexity. We also present a brief discussion of the issues analyzed. Table I summarizes our results and the formulas used in each case are available in the Appendix.

A. Approximate matching tool precision

The strategies compared in this paper were designed to work with ssdeep and sdhash; they were built considering the inner structure of each tool, leading the precision of a search more tied to the tool than the strategy itself. The latter only reduces the number of comparisons one should make and, therefore, speeds up the search process.

sdhash can detect resemblance and containment for a variety of object sizes without compromising its results. On the other hand, ssdeep is limited to only comparing objects of similar sizes and it is not suitable for dealing with large objects. Roussev [19] corroborates our statement showing that sdhash outperforms ssdeep in accuracy and scalability.

Considering the tools' precision aspect, the strategies using sdhash are a better choice than the ones using ssdeep, since the final result will be more accurate and scalable.

B. Memory requirement

Analysing Tab. I, we see the amount of memory each structure requires to work with a database of objects ranging from 1 GiB to 1 TiB, along with the compression rate. The details of our calculation are presented in the Appendix. All strategies have their own database to store the digests (Hash table or Bloom filters) except the Brute force, which may use external storage technology, as ordinary databases, files, xml etc (some may degrade the efficiency of the search).

Due to efficiency reasons, the structures should fit into main memory, a major problem for some strategies as the reference list grow. MRSH-NET, Brute force sdhash, and the BF-based tree would require loading bloom filters about 16, 25.6 and 336 GiB, respectively when working with a 1 TiB reference list. On the other hand, F2S2 would need only about 1.71

TABLE I
SIMILARITY SEARCH STRATEGIES: MEMORY REQUIREMENTS AND LOOKUP COMPLEXITY (SSDEEP VS. SDHASH)

Strategy	**Tools**	**Main technology**	**Memory requirements**				**Lookup complexity**
			1 GiB	**10 GiB**	**100 GiB**	**1 TiB**	
Brute force	sdhash	Bloom filters	25.60 MiB (2.50%)	256.00 MiB (2.50%)	2.50 GiB (2.50%)	25.60 GiB (2.50%)	$O(n)$
Brute force	ssdeep	Rolling Hash	0.19 MiB (0.02%)	1.87 MiB (0.02%)	18.75 MiB (0.02%)	192.00 MiB (0.02%)	$O(n)$
F2S2	ssdeep	Indexing + hash table	1.71 MiB (0.17%)	17.07 MiB (0.17%)	170.70 MiB (0.17%)	1.71 GiB (0.17%)	$O(n)$
MRSH-NET	sdhash, mrsh-v2	Single, huge Bloom filter	16.00 MiB (1.56%)	128.00 MiB (1.25%)	1.00 GiB (1.00%)	16.00 GiB (1.56%)	$O(1)$
BF-based tree	sdhash, mrsh-v2	Bloom filter + tree structure	176.00 MiB (17.19%)	1.79 GiB (17.90%)	17.64 GiB (17.64%)	336.00 GiB (32.81%)	$O(log(n))$

GiB of memory, a possible size to work with compared to the other approaches. The Brute force ssdeep would require even less memory, only 192 MiB. This enormous difference is mainly related to the size of the digest created by each tool: ssdeep produces digests of at most 96 bytes, while sdhash proportional to the object size ($\approx 2,6\%$).

Regarding memory requirement, ssdeep-based strategies proved to be more efficient and a better choice since they require less memory than the ones based on sdhash.

C. Lookup Complexity

The lookup complexity gives us an idea on how the strategies would scale with the increasing number of objects in the reference list. We presented in Table I the time complexity for a single query in a database of n objects. The best approach regarding this issue is MRSH-NET with $O(1)$ complexity, followed by the BF-based tree with $O(log(n))$, and then F2S2 and brute force with $O(n)$. Although MRSH-NET is limited to membership queries, its complexity is the lowest.

According to Winter, C. et al. [15], the time complexity of F2S2 can be split into two steps: Finding candidates that share the same n-gram of the queried item and calculating the similarity score between them. In the first case, we have a complexity of a $O(log(n))$ for fixed index table and $O(1)$ for dynamic resizing of the index table. In the second one, we have the complexity of a brute force search, since its effort is proportional to the size of the reference list. Summing the two steps, we get a complexity of $O(n) + O(log(n)) \approx O(n)$ for both dynamic resizing and fixed index table, when performing a single lookup. However, this complexity could not correspond to the real scenario in practice since the benefit of this strategy depends on the efficiency and effectiveness of the candidate's selection. Not all digests in such case will share the same n-grams as the queried item, so the number of comparisons will be restricted to only those that share. According to the authors' experiments, F2S2 is more than 2000 times faster than the brute force [15].

For this aspect, the sdhash-based strategies are the best choice, especially the Bloom filter-based tree that recovers a list of candidates just like F2S2 but with a lower time complexity. Although the lookup complexity is an important and necessary measurement to assess these approaches, in some cases having experiments evaluating the time spent are also a useful metric to measure the benefits of the strategy.

V. RESULTS

The approaches based on sdhash take advantage of the high precision in the detection of both resemblance and containment. They also presented a low lookup complexity, which is a good indicator for performing queries efficiently. However, they suffer from the same limitation: low compression rates. On the other hand, the ssdeep-based strategy (F2S2) showed low memory requirement, about 9 times smaller than MRSH-NET and 103 times smaller than the BF-based tree (1 TiB of data). However, it lacks in precision and has a high lookup complexity. Our findings show that none of the similarity search strategies fulfill the three aspects satisfactorily since each approach is better in one aspect with respect to the others. The desirable strategy would be the one presenting the precision of BF-based tree, the memory requirement of F2S2 and the lookup complexity of MRSH-NET. As future work, we will study other strategies that can better address all three requirements in order to come up with a better solution to the similarity search problem.

VI. CONCLUSIONS

Known File Filtering is an efficient method for performing white/black-listing. Due to limitations on hash functions, approximate matching schemes arrive as a good, but expensive, solution to mitigate such constraints. However, the major bottleneck in forensics investigations using this filtering approach is not the digest generation, but the similarity search. In this paper, we compared some similarity search strategies based on the two best known approximate matching tools. We showed that none of the strategies address well the three aspects discussed: Tool's precision, memory requirement, and lookup complexity. Future studies encompass comparing other approaches, expanding the characteristics compared, and analyzing other approximate matching tools.

REFERENCES

[1] NIST. National software reference library. http://www.nsrl.nist.gov, 2016. Accessed 2016 Set 13.

[2] Frank Breitinger, Barbara Guttman, Michael McCarrin, Vassil Roussev, and Douglas White. Approximate matching: definition and terminology. *NIST Special Publication*, 800:168, 2014.
[3] Andrew Tridgell. *Efficient algorithms for sorting and synchronization.* Australian National University Canberra, 1999.
[4] Andrew Tridgell. Spamsum. *URL http://samba.org/ftp/unpacked/junkcode/ spamsum/README*, 2002.
[5] Jesse Kornblum. Identifying almost identical files using context triggered piecewise hashing. *Digital investigation*, 3:91–97, 2006.
[6] Landon Curt Noll. FNV hash, 2001. Accessed 2017 Apr 18.
[7] L. Chen and G. Wang. An efficient piecewise hashing method for computer forensics. In *First International Workshop on Knowledge Discovery and Data Mining (WKDD 2008)*, pages 635–638, Jan 2008.
[8] Frank Breitinger and Harald Baier. *Performance Issues About Context-Triggered Piecewise Hashing*, pages 141–155. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.
[9] Harald Baier and Frank Breitinger. Security aspects of piecewise hashing in computer forensics. In *IT Security Incident Management and IT Forensics (IMF), 2011 Sixth International Conference on*, pages 21–36. IEEE, 2011.
[10] Vassil Roussev. Data fingerprinting with similarity digests. In *IFIP International Conf. on Digital Forensics*, pages 207–226. Springer, 2010.
[11] Vassil Roussev. Building a better similarity trap with statistically improbable features. In *System Sciences, 2009. HICSS'09. 42nd Hawaii International Conference on*, pages 1–10. IEEE, 2009.
[12] Frank Breitinger, Harald Baier, and Jesse Beckingham. Security and implementation analysis of the similarity digest sdhash. In *First international baltic conference on network security & forensics (nesefo)*, 2012.
[13] Jonathan Oliver, Scott Forman, and Chun Cheng. Using randomization to attack similarity digests. In *International Conference on Applications and Techniques in Information Security*, pages 199–210. Springer, 2014.
[14] Donghoon Chang, Somitra Kr Sanadhya, Monika Singh, and Robin Verma. A collision attack on sdhash similarity hashing. In *Proceedings of 10th intl. conference on systematic approaches to digital forensic engineering*, pages 36–46, 2015.
[15] Christian Winter, Markus Schneider, and York Yannikos. F2s2: Fast forensic similarity search through indexing piecewise hash signatures. *Digital Investigation*, 10(4):361–371, 2013.
[16] Frank Breitinger, Harald Baier, and Douglas White. On the database lookup problem of approximate matching. *Digital Investigation*, 11:S1–S9, 2014.
[17] Frank Breitinger and Harald Baier. *Similarity Preserving Hashing: Eligible Properties and a New Algorithm MRSH-v2*, pages 167–182. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.
[18] Frank Breitinger, Christian Rathgeb, and Harald Baier. An efficient similarity digests database lookup-a logarithmic divide & conquer approach. *The Journal of Digital Forensics, Security and Law: JDFSL*, 9(2):155, 2014.
[19] Vassil Roussev. An evaluation of forensic similarity hashes. *Digital investigation*, 8:34–41, 2011.

APPENDIX

To compare the strategies regarding the memory required to load and work with them, we developed and adapted some formulas.

A. Brute force cost

To estimate the memory requirement for this strategy, we need to consider the approximate matching tool being used, since some of them present some singularities. For ssdeep, the memory estimate is given by Eq. 1.

$$m_{ss} = n \cdot s_{ss} \ (bits) \tag{1}$$

where n is the number of objects in the reference list and s_{ss} is the size of the digest created by ssdeep in bits.

For sdhash, we first need to know the number of features presented by an object (in average). Breitinger, F. et al. [18] say that *"sdhash maps 160 features into a bloom filter for every approximately 10 KiB of the input file"*. This way, we can calculate z (number of features) by Eq. 2.

$$z = (\mu \cdot 2^{20} \cdot 160)/(10 \cdot 2^{10}) = 2^{14} \cdot \mu \tag{2}$$

where μ is the reference list size (MiB) and 2^{20} and 2^{10} are factors to change from MiB and KiB to bytes, respectively. Now we calculate how many Bloom filters are necessary to represent all features and hence the memory requirements for sdhash, using Eq. 3.

$$m_{sd} = (z \cdot s_{bf})/f_{max} \ (bits) \tag{3}$$

where s_{bf} is the size of each bloom filter (bits) and f_{max} the maximum number of features allowed to be inserted by sdhash in each filter.

B. F2S2 cost

We can estimate the amount of memory required by Eq. 4:

$$m_{F2S2} = n \cdot s_{ss} \cdot (1 + p_{fac}) + s_{name} \ (bytes) \tag{4}$$

where n is the number of objects in the reference list, s_{ss} digests length, p_{fac} the payload factor added for the index (between $7-8$) and s_{name} the length of the objects names in the reference list.

C. MRSH-NET cost

From Breitinger's work [16] we can calculate the amount of memory required using Eq. 5.

$$m_{MRSH} = \frac{k \cdot s \cdot 2^{14}}{ln(1 - \sqrt[k \cdot r_{min}]{p_f})} \ (bits) \tag{5}$$

where k is the number of sub-hashes, s the object set size (MiB), 2^{14} the number of features per set s, r_{min} the minimum number of consecutive features and p_f the probability of false positive for an object.

D. BF-based tree cost

To estimate the amount of memory required for the BF-based tree structure, we can use Breitinger's equations [16]. First, we need to determine the size of the root bloom filter, given by Eq. 6:

$$m_{BFroot} = -z \cdot \frac{ln\, p}{(ln\, 2)^2} \ (bits) \tag{6}$$

where z is the number of features in the whole set and p the false positive probability for a feature, calculated by: $p = \sqrt[r_{min}]{p_f}$. The parameter r_{min} is the number of consecutive features and p_f the false positive probability for an object.

Next, we need to calculate the level of the tree using Eq. 7:

$$h = log_x(n), \tag{7}$$

where x is the degree of the tree (e.g. $x = 2$ for a binary tree).

Finally, we estimate the memory required for the Bloom filter tree structure by Eq. 8:

$$m_{BFtree} = m_{BFroot} \cdot h \ (bits) \tag{8}$$

where m_{BFroot} is the size of the root bloom filter given by Eq. 6 and h the level of the tree (Eq. 7).