Position papers of the Federated Conference on Computer Science and Information Systems, pp. 51–57
DOI: 10.15439/2017F535
ISSN 2300-5963 ACSIS, Vol. 12

FLOUDS: A Succinct File System Structure

Daniel Peters*, Johannes Fischer†, Florian Thiel* and Jean-Pierre Seifert‡
*Physikalisch-Technische Bundesanstalt (PTB), Germany
Email: {daniel.peters, florian.thiel}@ptb.de
†Department of Computer Science, TU Dortmund, Germany
Email: johannes.fischer@cs.tu-dortmund.de
‡Security in Telecommunications, TU Berlin, Germany
Email: jpseifert@sec.t-labs.tu-berlin.de

***Abstract*—To spot malicious manipulation, remote attestation and maintenance for devices that are under legal control is very important. One example are measuring instruments, where the manufacturer and the market surveillance want to check if system integrity is preserved. In Europe, legal requirements state that a software identifier needs to be supplied/output by the device, which is often just a checksum over the files that are considered to be legally relevant for the measuring purpose. As measuring instruments and also other legally monitored devices are often small embedded systems, the need for a fast algorithm arises that creates a small file system list containing as much information as possible. In this paper, a new file system structure called FLOUDS is explained that fulfills these requirements. The FLOUDS uses theoretical optimal space to represent the file system structure, while it, nevertheless, enables fast file searches by names and also properties. For example, all files of a specific file type, e.g., pictures, movies, executables, etc., can be listed in $O(p \lg n)$ time, where p is the number of files of the specific file type searched for, and, where n represents the total number of file types in the system.**

I. Introduction

IN MANY states, law obliges manufacturers of devices that are under legal control, to implement an easy procedure to output a software identifier. For example, measuring instruments under legal control, e.g., commodity meters for the supply of gas, water and electricity, etc., output these identifiers to show market surveillance agents that the approved software is still running on the device and system integrity is preserved. Hereby, the agents check the devices on sidein predifined intervals, e.g. every two years. Often, just a checksum over the legally relevant files is calculated. As measuring instruments and also other legally supervised devices are ofen powerful embedded instruments, it can be inferred that more efficient algorithms can be implemented that enhance the checksum or hash value with additional information, like the file system structure. Additionally, data exchange between devices over the internet has become an important aspect, nowadays. In the era of the Internet of Things (IoT), the number of these devices will, according to Gartner [46], exceed 25 billion in the year 2020. As storage units have become smaller and cheaper, these devices can already save millions of files. Considering that in the future the automatic data exchange between these devices will blossom, the creation of a small data structure listing all the files on a device is handy. Despite being small, this data structure should also be quickly traversable. It should list as much information about a file as possible to check, for example, the name, the format, the size, and the checksum.

This paper describes such a data structure which makes use of succinct approaches to store trees. In this structure, a fast file search is made possible by using space-efficient algorithms to store the file names. Hence, the data structure is not only usable as a file list, but can easily be used as the fundamental structure for a read-only file system in which files can be located and listed efficiently.

A. Outline

The paper is structured as follows: In Section I, an introduction about the topic will be given, outlining the importance and usability of a succinct file system structure. In Section II, an overview of succinct data structures will be supplied, explaining important operations like *rank*- and *select*, which are needed to traverse many succinct data structures, one such data structure is the "Level Order Unary Degree Sequence" (LOUDS). The "File system Level Order Unary Degree Sequence" (FLOUDS), which is based on the LOUDS, will be explained in detail in Section III. Afterwards, practical tests in Section IV will show the efficiency of the FLOUDS by comparing it with the *locate* database of UNIX systems. At the end, before the conclusion in Section VI, a discussion is given in Section V which describes where the FLOUDS can be used and how file modifications can be handled.

II. Succinct Data Structures

In this section existing data structures are presented that form the basis of the new succinct file system representation. All the results are in the word-RAM model of computation, i.e. the machine consists of words of width w bits that can be manipulated in $O(1)$ time by a standard set of logical and arithmetic operations, and the problem size n is not larger than $O(2^w)$.

In the past two decades, succinct data structures have been one of the key contributions to the algorithmic community. The aim of these structures is to represent objects from a universe of size u in information-theoretical optimal space $\lg u$ bits of space (function lg denotes the binary logarithm throughout this paper). Additionally, fast operations should be supported, ideally in time no worse than with a "conventional" data structure for the object. Usually, a space overhead of no more than $o(\lg u)$ bits space arises for this property.

Ordered trees are just one example, where succinct data structures yield good result. Hereby, with n nodes we have a universe of size $u \approx 4^n$. In 1989, Jacobson first described such a tree representation that used only $10n + o(n)$ bits, while supporting the most common navigational operations in $O(\lg n)$ time [29]. With time, new succinct data structure for trees where developed that use the optimal $2n + o(n)$ bits and optimal $O(1)$ navigation time, e.g. [39]. A conventional, pointer-based data structure, for example, requires $\Theta(n \lg n)$ bits.

There are many more examples of succinct data structures: bit-vectors [41], dictionaries [40], binary relations [3], permutations [37], suffix trees [45], etc. In nearly all cases, attempts were made at practical implementations, with successful results [20], [24], [31].

A. Rank and Select

Many succinct data structures make use of two fundamental operations, called *rank-* and *select*. In this paper, these operations on S, with $S[1, n]$ being a *bit-string* of length n, are defined as follows:

- $\mathsf{rank}_1(S, i)$ gives the number of 1's in the prefix $S[1, i]$
- $\mathsf{select}_1(S, i)$ gives the position of the i'th 1 in S, reading S from left to right $(1 \leq i \leq n)$

To give an example, in a string $S = 01001$ of size 5, where position 1 denotes the leftmost bit, $\mathsf{rank}_1(S, 3) = 1$, and $\mathsf{select}_1(S, 2) = 5$. Operations $\mathsf{rank}_0(S, i)$ and $\mathsf{select}_0(S, i)$ are defined similarly for 0-bits. S can be represented in $n + o(n)$ bits such that rank- and select-operations are supported in $O(1)$ time [39].

The approach used to achieve this is to divide the bit-string S into blocks of fixed size, e.g. 64 bit, and store the number of ones before the blocks. These blocks can be combined into bigger blocks, e.g. of size 4096 bits, often called super-blocks, which again store the number of ones at the position in front of each super-block. After each super-block the number of ones for the upcoming smaller block is reset to 0. Every super-block needs $\lg n$ bits and with the example of 4096 bit size super-blocks, each small block needs only $\lg(4096) = 12$ bits to store the number of ones till this position. $\mathsf{rank}_1(S, i)$ can then be performed by accessing the blocks and adding them together. Getting the number of ones inside a block is done by bit-operations and/or table-lookups to speed up the process. The approach for $\mathsf{select}_1(S, i)$ is similar but a little bit trickier, a nice description can be found in [9], where a three level directory structure is used.

B. Storing Trees Succinctly

There are several ways to represent an ordered tree with n nodes using $2n$ bits. The best known are the "level order unary degree sequence" (LOUDS), the "balanced parantheses" (BP) and the "depth first unary degree sequence" (DFUDS) [5], [29], [38]. A comparison of these structures is depicted in Figure 1.

The LOUDS is formed by performing a breadth-first traversal (BFT) on the tree, starting with an artificial super-root that is added in front of the real root node and connected to it. At every step of the BFT 1^d0 is added to the LOUDS for each node, with d being the number of children of the respective node.

The BP and DFUDS use the depth-first traversal (DFT) for their construction. The BP is constructed as follows: when the DFT descends a level, an opening parenthesis is written, and when it ascends, a closing one is written.

The DFUDS combines the BP and the LOUDS. Hereby, when the DFT descends to a node, d open parentheses are written out, with d being the number of the children of that node. After the node traversal a closing parenthesis is written out. Like in the LOUDS, an opening parentheses is added at the beginning, similar to the artificial super-root.

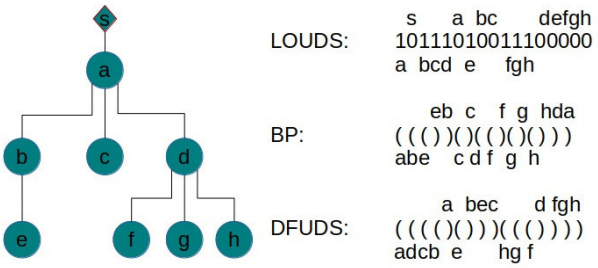

Fig. 1: Comparison between LOUDS, BP und DFUDS (s is the artificial super-root)

For the LOUDS only *rank-* and *select* is needed to enhance the data structure with navigational operations. The BP and the DFUDS need some more structures:

- enclose(S, p): Finds the pair of parentheses, which encloses the open parentheses at position p most tightly and returns the position of the open parenthesis of the pair of parentheses.
- findclose(S, p): Returns the position of the closing parentheses which belongs to the open parentheses at position p.
- findopen(S, p): Returns the position of the open parentheses which belongs to the closing parentheses at position p.

All these functions can be implemented in $O(1)$ time with $o(n)$ space, with very small and fast practical data structures, see, e.g [19]. As can be noticed, all succinct data structures for trees [5], [10], [14], [29], [38] must have the freedom to fix a particular naming for the nodes; natural such namings are post- or pre-order [5], [29], [38], in-order [10], and level-order [29].

As the LOUDS is easier to implement and needs only *rank-* and *select*, it practically also uses less space. Therefore, we think the best choice is to use it as the fundamental structure of the file system structure explained in Section III.

Augmenting the LOUDS with *rank-* and *select* results in the total space of $2n + o(n)$ bits, where the basic navigational operations on trees are simulated in $O(1)$ time: Getting the parent of node i $(1 \leq i \leq n)$ is done by jumping to the

position y of the i'th 1-bit in S by $y = \mathsf{select}_1(S, i)$, and then by counting the number j of 0's that are present before y, with $j = \mathsf{rank}_0(S, y)$. The resulting j represents the level-order number of the parent of i. Listing the children of i is done by going to the position x of the i'th 0-bit in S by $x = \mathsf{select}_0(S, i)$, and then iterating over the positions $x + 1, x + 2, \ldots$, as long as the corresponding bit is '1'. For each such position $x+k$ with $S[x+k] = 1$, the level-order numbers of i's children are $\mathsf{rank}_1(S, x) + k$, which can be simplified to $x - i + k + 1$.

To give an example, we look more close at Fig. 1. Here, the LOUDS is $S = 10111010011100000$, with the corresponding tree depicted at the left hand side of the figure. Now, to get the children of the forth node (in Fig. 1 denoted as d), we calculate the position of the first child of d in S: $x_1 = \mathsf{select}_0(S, 4) + 1 = 10$. We then check if $S[x_1] = 1$ and if so ($S[10] = 1$) we increment the position by one until we arrive at a 0. In our example until position 13, i.e. positions 10, 11, 12 are the positions of the children of node 4 (d). To convert the positions tho the tree numbers, we then have to calculate $c_1 = \mathsf{rank}_1(S, 10) = 6$ (f in alphabetical numbering as shown in Fig. 1) , $c_1 = \mathsf{rank}_1(S, 10) = 7$ (g), and $c_1 = \mathsf{rank}_1(S, 10) = 8$ (h). For a parent, let us take node the third node as an example (in Fig. 1 denoted as c). Here, $y = \mathsf{select}_1(S, 3) = 4$ is the position of the node in the LOUDS, and with $j = \mathsf{rank}_0(S, 4) = 1$, we get 1 as a result ($a$).

C. Wavelet Trees

The operations *rank-* and *select* have been extended to sequences over larger alphabets, at the cost of slight slowdowns in the running times [4], [21]. In this section a practical approach is discussed, called *wavelet tree* [23]. A wavelet tree is constructed as follows: First each character c in a text S is assigned to exactly one bit (a 0 or a 1). The root node v_1 is situated on the first level and contains the bit-vector B_1 and the actual text $S_1 = S$. Now the tree is built recursively: If a node v contains a text S_v that has at least two different characters, then two child nodes v_l and v_r are created. All characters which are marked with a 0 go to the left node and all other characters go to the right node. Note that at the end the S_v's of every node are not saved, only their bit vectors B_v with the *rank-* and *select* data structures, and the mappings "c to leaf" and "leaf to c". If a balanced wavelet tree is constructed, in which the first half of a node's alphabet is written into the left child and the other half into the right one, the mappings from "c to leaf" and from "leaf to c" do not need to be stored, and the tree can still be easily traversed. Figure 2 shows such a balanced wavelet tree for $S = 303302013032012010010$.

The advantage of these wavelet trees is that $\mathsf{select}_c(S, i)$, $\mathsf{rank}_c(S, j)$ and access(j) queries for an alphabet of size σ can be answered in $O(\lg \sigma)$ time for every character c in S and position j in S, while using only $n \lg \sigma + o(n \lg \sigma)$ space.

To give an example, we look at Fig. 2. Here, $\mathsf{rank}_3(S_1, 6)$ can be calculated as follows. We know that 3 is in the second half of our alphabet (0, 1 are represented as a 0 in B_1, and the numbers 2 and 3 are represented as a 1). So first,

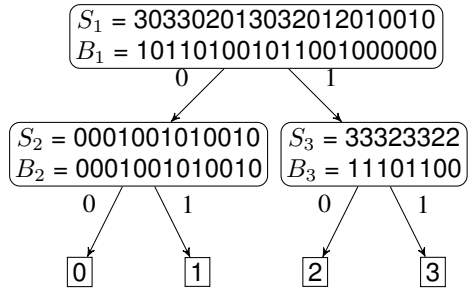

Fig. 2: Illustration of a balanced wavelet tree.

$\mathsf{rank}_1(B_1, 6) = 4$ and then $\mathsf{rank}_1(B_3, 4) = 3$ (here again 3 is represented as a 1 in B_2 because it is in the second half of the alphabet, and 2 in the first), meaning in total $\mathsf{rank}_3(S_1, 6) = 3$. For select a similar procedure is being used only that now we start from the leaves. To calculate $\mathsf{select}_1(S_1, 2)$, we know that 1 is represented by a 0 in B_1 and as a 1 in B_2. Hence, we get $\mathsf{select}_1(B_2, 2) = 7$, and afterwards $\mathsf{select}_0(B_1, 7) = 14$, so the result is $\mathsf{select}_1(S_1, 2) = 14$.

III. FLOUDS

An approach that is also based on the LOUDS to store tree-like graphs, which file systems with links can be regarded as, is explained in [16]. This method was developed primarily for storing phylogenetic networks (phylogenetic networks are used in biology to express relationships between species). The structure consists of a trit (ternary digit) variant to store the enhanced LOUDS, which is rather not suitable for file systems, because one can only differentiate between two types of files, e.g., links and regular files (the 0s in the trit-variant are used to end the children listing of a node as in the LOUDS). In this section, a new version called FLOUDS ("File system Level Order Unary Degree Sequence") is described, which makes use of the wavelet tree presentation described in Section II-C. Therefore, files can be divided into more than two types, e.g. the types listed in Figure 3:

- regular files
- directories
- links (hard-, soft)
- block-oriented device
- char-oriented device

This list can be expanded, e.g. to sockets, pipes, MIME types (pictures, videos, ...) etc., or shortened as needed.

The FLOUDS is created as follows: First a BFT at the root of the file system tree is started. For every node (file/folder) a predefined number representing the file type is appended to S. If the node is the first child of its father, a 1 is written to B, otherwise a 0. The array S can be created directly as a wavelet tree, when the number of file types t is known from the beginning; if not, it can be created later, after the complete BFT run.

Additionally, the file/folder names are successively written to N, whereas B_n marks the beginning of a new file name by a 1. After a complete BFT S, B, N and B_n are created and

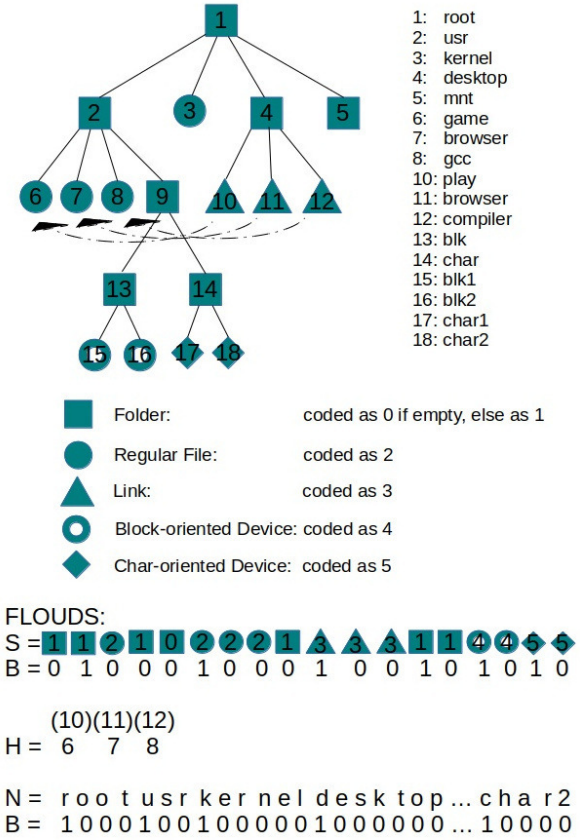

Fig. 3: File system structure FLOUDS, consisting of S, B, H, N and B_n.

the number of nodes n and the number of links l are known. This information can then be used to create the wavelet tree H, which is needed to retrieve the link address to the file. In total H requires $l \lg n + o(l \lg n)$ space. In H the FLOUDS numbers of the real files of the links are stored, in the order in which they were traversed by the BFT. Now, H can be used to check if a file is a link, and afterwards get the linked file, as described in the next section.

A. Navigating through the FLOUDS

With the help of S and B, a file system tree traversal can be done much like it in the LOUDS. Functions *parent* and *child* are as follows:

- $\text{child}(x, i) = \mathsf{select}_1(B, \mathsf{rank}_1(S; x)) + i - 1$, if $S[x] = 1$ and $i <$ number of children
- $\text{parent}(x) = \mathsf{select}_1(S, \mathsf{rank}_1(B; x))$

For example, in Figure 3, the folder-entries of node 9 can be listed by first checking if node 9 is a folder $S[9] = 1$, and then getting its folder-number $f_n = \mathsf{rank}_1(S; 9) = 4$. Afterwards, a jump to its first child is done, $\text{child}(9, 1) = \mathsf{select}_1(B; 4) = 13$. All the children of node 9 are between the first and the last child l: $\text{child}(9, l) = \mathsf{select}_1(B; 5) - 1 = 14$ (the position before encountering the next '1': $4 + 1 = 5$). So the folder-entries of node 9 are 13 and 14 (and $l = 2$).

The time complexity of outputting a child (folder entry) lies in $O(\lg t)$, with t being the number of file types in the FLOUDS, because all rank and select operations on B are supported in $O(1)$, and for S, which is saved in a wavelet tree format, in $O(\lg t)$.

The array H can be used to check if a node is a link (in the example from Figure 3: $S[n] = 3$) or if a file is referenced by links, and where these links are situated in the file system tree, by using the wavelet tree of H:

- $\text{getLink}(n, i) = \mathsf{select}_3(S; \mathsf{select}_n(H; i))$
- $\text{getOrig}(n) = H[\mathsf{rank}_3(S; n)]$, if $S[n] = 3$

Looking again at the example of Figure 3: the first link that points to node 7 is $\text{getLink}(7, 1) = \mathsf{select}_3(S; \mathsf{select}_7(H; 1)) = 11$; and the file-number (node) 12 points to is $\text{getOrig}(12) = H[\mathsf{rank}_3(S; 12)] = 8$. Hence, the FLOUDS also enables to print out its direct parent directory, and additionally, if it is a link or has links pointing to it, the parent directories the other links are stored in. Hereby, the time complexity depends on the wavelet tree of H and S to find the files, so summing up, listing all parent folders of a file including the parent folders of its links, is in $O(a*(\lg l + \lg t))$, with l being the number of all original files that have links, t the number of total file types in the whole file system, and a the actual number of links of the searched file.

Additionally, with the help of *rank*- and *select*, the following functions are directly executable:

- Getting the number of files in a folder and listing them.
- Getting the number of files of a specific file-type in a folder and listing them.

For example, if every file is assigned a MIME-type number, all pictures in the file system can be listed efficiently.

B. Efficiently Storing File Names

A simple method to find files fast can be achieved by sorting the folder entries in alphabetical order. Hereby, the prefix of a file name in a folder can be found by a binary search.

A more efficient search that also finds substrings of file names can be achieved by applying 2-Way dictionaries. An example would be using the Burrows-Wheeler Transform (BWT) [8] with Run-Length Encoding. Such a method is described in [15], for example. Hereby, substrings can be found efficiently, and afterwards be mapped to their FLOUDS numbers, or the file names can be read out via the FLOUDS numbers, respectively.

Another method that aims at compression, is described in [2]. There, the Lempel-Ziv algorithm 78 (LZ78 [47]) is altered in a way that makes locating prefixes possible. The LZ78 compression algorithm for a string $S[1, n]$ proceeds by parsing S from left to right. Hereby, S is divided into blocks that are one-letter extensions of previously parsed substrings. The set of current blocks is called the phrase dictionary D. The dictionary D is prefix-closed and represented with a trie (the LZ-trie), which is stored in [2] with some enhancements to achieve look-up and access support. The method is good for checking the integrity of file system structures, because finding

substrings of file names is not really needed (the exact file name is known a priori).

C. Integrity checking

There are several variations to check file system integrity on request:

1) The FLOUDS is signed and transferred with the file names.
2) The FLOUDS is newly created and while traversing the file system tree, a hash value of each file is calculated. These hash values are written into a hash array of size n and transmitted together with the FLOUDS and the file names.
3) To save space, the file name list can be omitted. The file names can just be included in the computation of the hash values, mentioned in point 2.
4) Only a predefined number of files are hashed to save even more space.
5) Only one hash value over the entire structure is calculated and transmitted.

The fifth method requires the least amount of space, because only a hash value needs to be transmitted. This value can, for example, be displayed on the device's display. If the hash has an unexpected value, one of the other four methods can be executed to check, how the file system structure has changed and which files have been altered.

The used hash algorithm should be as collision-free as possible. Still, it can be freely chosen, for example from a simple checksum like CRC16, to secure hashing algorithms such as SHA-2, depending on performance, space or safety demands.

IV. Practical Results

The aim of this section is to show the practicality of our approach by comparing it with a well known database for Unix systems, which the *locate* command uses to efficiently find files.

For our test we used the succinct libraries https://github.com/ot/succinct and https://github.com/simongog/sdsl, which have well-tuned succinct data structure implementations (other sources are [1], [18]). Our machine was equipped with an Intel Core i7@2.2GHz and 8GB of RAM, running under Ubuntu 14.04.

Table I shows the sizes of the *mlocate* database, which in Unix systems is normally situated at "/var/lib/mlocate/mlocate.db", the size of the FLOUDS with the file names just stored in plain text, and the lzFLOUDS with the file names stored by the LZ78 method [2] described in Section III-B.

We used 4 different file lists for comparison:

1) buildroot: buildroot (http://buildroot.uclibc.org/) is a tool to generate embedded Linux systems, the file system we generated contained 435 nodes (files/folders/link etc.).
2) linux_src: The Linux 4.2 Kernel source tree containing 54 171 nodes.
3) comp1: A small file system of a desktop computer containing 333 854 nodes.
4) comp2: A bigger file system of another desktop computer with 1 853 354 nodes.

TABLE I: Comparison of sizes in MB: 1. buildroot, 2. linux_src, 3. comp1, 4. comp2.

F	mlocate	FLOUDS	lzFLOUDS
1.	0.004	0.003	0.002
2.	0.957	0.625	0.218
3.	7.722	4.964	1.932
4.	42.876	22.928	9.141

The second table, Table II, compares the running times to find a file name. Hereby, the LZ78 variant of the FLOUDS searched for exact matches, whereas the other two searched for substrings. We averaged the running times over 100 tests, searching for random strings.

TABLE II: Comparison of running times in sec: Legend as in Table I.

F	mlocate	FLOUDS	lzFLOUDS
1.	0.004	0.027	$8 * 10^{-6}$
2.	0.035	0.105	$10 * 10^{-6}$
3.	0.381	0.565	$9 * 10^{-6}$
4.	0.823	9.854	$12 * 10^{-6}$

It can be observed that the naive FLOUDS representation is already around 40% smaller than the *mlocate* database. Still the running times for file-systems with many files are up to 12 times slower. On the other hand, the lzFLOUDS is some magnitudes faster than the other structures. The drawback of the lzFLOUDS is that it can only be used to find prefixes, and in our tests it just output a result, if the exact match was found. We think that this is enough, if the FLOUDS is used for integrity checking, because the exact file names should be known.

V. Discussion

Succinct data structures perform very well on static objects and not that well on dynamic ones. The same applies to the FLOUDS, hence it is better suited for a read-only file system. For many embedded devices this poses no restriction, because mostly they are put in commission to fulfil only a certain scope; changes to the file system structure are often not allowed and would be a sign of malicious manipulation. However, if changes are to be made, the FLOUDS needs to be rebuilt. A fast rebuild can be easily achieved if the names are not compressed and saved in plain form. For the other representations, the file name string needs to be completely rebuilt, which can be slow. Still, dynamic succinct data structures exist that can be used to this end, e.g., [34].

Figure 4 shows how the file system manager should be separated from the computing component to hinder malicious applications from tampering with files if file integrity is an

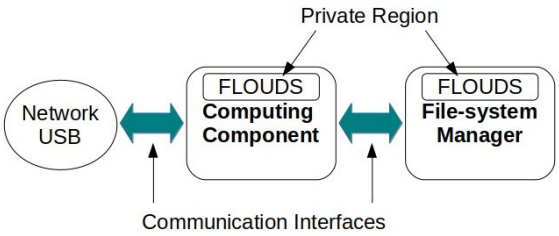

Fig. 4: Separating the file system Manager

issue. These components can be separate devices, where the file system component is at a secure location. Hereby, the computing component could have a copy of the FLOUDS in its internal storage, to speed up locating files. If a file is needed, the FLOUDS-number and the number of requested bytes can be sent to the file system manager, which then answers the request. This communication can be encrypted if an open network is used for communication. In the same manner, an entitled entity can check the integrity of the file system by just sending a request to the computing component, which in turn gets the FLOUDS encrypted with the private key of the file system manager. Afterwards, it sends this encrypted FLOUDS back to the entitled authority. After decrypting it with the public key of the file system component, the entitled authority can make sure that the computing component did not manipulate the FLOUDS-file (assuming of course that the private key of the file system manager is not known by the computing component).

Another approach is software-separation through virtualization. There are some papers that come to the conclusion that the microkernel approach for virtualization is recommended to construct secure systems [26]–[28], [33], [35], [42]. A concrete modular system architectures is described in [43], which yields good results also for embedded devices. It is based on virtualization by dividing critical parts of software from non-critical ones through virtual machines.

Another area of application for the FLOUDS is to use it as the fundamental structure of a whole file system. A possible approach would be to combine it with a read-only, compressed file system like squashFS (http://squashfs.sourceforge.net/). The idea is to use the FLOUDS numbers of the nodes to represent the block numbers, which in turn point to compressed blocks. Nevertheless, if compression is not that important, the FLOUDS needs to be evaluated against a dynamic file system, e.g., based on the B^ϵ-tree, because these show good results for locating files and have fast write operations [11], [30].

VI. Conclusion

In this paper, a flexible file list structure called FLOUDS was presented. This structure was explained in detail to show that it can be used in many ways. Firstly, it is well suited for embedded devices that need to be validated for integrity in commission, or which want to exchange file lists. Secondly, it can be used as the groundwork for a read-only file system, which uses as little space as possible. Lastly, it can be used to efficiently find and list files by using succinct data structures for trees and 2-way dictionaries. For the latter, the FLOUDS was evaluated by comparing it to another file name database, which is used in Unix systems by the *locate* command. These tests show that the FLOUDS stores more information, is smaller, and for integrity checking also faster.

References

[1] D. Arroyuelo, R. Cánovas, G. Navarro, and K. Sadakane. Succinct trees in practice. In *Proc. ALENEX*, pages 84–97. SIAM, 2010. https://doi.org/10.1137/1.9781611972900.9

[2] J. Arz and J. Fischer. LZ-Compressed String Dictionaries. In *Data Compression Conference (DCC)*, pages 322 – 331, 2014. https://doi.org/10.1109/DCC.2014.36

[3] J. Barbay, F. Claude, and G. Navarro. Compact rich-functional binary relation representations. In *Proc. LATIN*, volume 6034 of *LNCS*, pages 170–183. Springer, 2010. https://doi.org/10.1007/978-3-642-12200-2_17

[4] D. Belazzougui and G. Navarro. New lower and upper bounds for representing sequences. In *Proc. ESA*, volume 7501 of *LNCS*, pages 181–192. Springer, 2012. https://doi.org/10.1007/978-3-642-33090-2_17

[5] D. Benoit, E. D. Demaine, J. I. Munro, R. Raman, V. Raman, and S. S. Rao. Representing trees of higher degree. *Algorithmica*, 43(4):275–292, 2005. https://doi.org/10.1007/s00453-004-1146-6

[6] D. K. Blandford, G. E. Blelloch, and I. A. Kash. Compact representations of separable graphs. In *Proc. SODA*, pages 679–688. ACM/SIAM, 2003. http://doi.acm.org/10.1145/644108.644219

[7] D. K. Blandford, G. E. Blelloch, and I. A. Kash. An experimental analysis of a compact graph representation. In *ALENEX/ANALC*, pages 49–61. SIAM, 2004. https://doi.org/10.1109/BOD.2006.320815

[8] M. Burrows and D. Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.

[9] D. Clark. Compact Pat Trees. Phd Thesis presented to the University of Waterloo, Canada. 1996.

[10] P. Davoodi, R. Raman, and S. R. Satti. Succinct representations of binary trees for range minimum queries. In *Proc. COCOON*, LNCS, pages 396–407. Springer, 2012. https://doi.org/10.1007/978-3-642-32241-9_34

[11] J. Esmet, M. A. Bender , M. Farach-Colton, and B. C. Kuszmaul. The TokuFS streaming file system. In *Proceedings of the 4th USENIX Workshop on Hot Topics in Storage (HotStorage)*, 2012.

[12] A. Farzan and J. Fischer. Compact representation of posets. In *Proc. ISAAC*, volume 7074 of *LNCS*, pages 302–311. Springer, 2011. https://doi.org/10.1007/978-3-642-25591-5_32

[13] A. Farzan and J. I. Munro. Succinct representation of arbitrary graphs. In *Proc. ESA*, volume 5193 of *LNCS*, pages 393–404. Springer, 2008. https://doi.org/10.1007/978-3-540-87744-8_33

[14] A. Farzan and J. I. Munro. A uniform approach towards succinct representation of trees. In *Proc. SWAT*, volume 5124 of *LNCS*, pages 173–184. Springer, 2008. https://doi.org/10.1007/978-3-540-69903-3_17

[15] P. Ferragina and R. Venturini. The compressed permuterm index. In *ACM Trans. Algorithms 7, 1*, Article 10, 21 pages, 2010. https://doi.org/10.1145/1277741.1277833

[16] J. Fischer and D. Peters. A Practical Succinct Data Structure for Tree-Like Graphs. In *WALCOM: Algorithms and Computation*, pages 65–76, Springer, 2015. https://doi.org/10.1007/978-3-319-15612-5_7

[17] C. Gavoille and N. Hanusse. On compact encoding of pagenumber k graphs. *Discrete Mathematics & Theoretical Computer Science*, 10(3):23–34, 2008.

[18] R. F. Geary, N. Rahman, R. Raman, and V. Raman. A simple optimal representation for balanced parentheses. *Theor. Comput. Sci.*, 368(3):231–246, 2006. https://doi.org/10.1007/978-3-540-27801-6_12

[19] S. Gog and J. Fischer Advantages of Shared Data Structures for Sequences of Balanced Parentheses. In *Proceedings of the 2010 Data Compression Conference (DCC'10)*, IEEE Press, pages 406–415, 2010. http://doi.org/10.1109/DCC.2010.43

[20] S. Gog and E. Ohlebusch. Fast and lightweight LCP-array construction algorithms. In *Proc. ALENEX*, pages 25–34. SIAM, 2011. http://dx.doi.org/10.1137/1.9781611972917.3

[21] A. Golynski, J. I. Munro, and S. S. Rao. Rank/select operations on large alphabets: a tool for text indexing. In *Proc. SODA*, pages 368–373. ACM/SIAM, 2006. http://dx.doi.org/10.1145/1109557.1109599

[22] R. González, S. Grabowski, V. Mäkinen, and G. Navarro. Practical implementation of rank and select queries. In *Poster Proceedings Volume of 4th Workshop on Efficient and Experimental Algorithms (WEA)*, pages 27–38, Greece, 2005. CTI Press and Ellinika Grammata. http://dx.doi.org/10.1007/978-3-540-89097-3_18

[23] R. Grossi, A. Gupta, and J. S. Vitter High-order entropy-compressed text indexes. In *Proceedings of the 14th Annual SIAM/ACM Symposium on Discrete Algorithms (SODA)*, pages 841–850. 2003.

[24] R. Grossi and G. Ottaviano. Design of practical succinct data structures for large data collections. In *Proc. SEA*, volume 7933 of *LNCS*, pages 5–17. Springer, 2013. http://dx.doi.org/10.1007/978-3-642-38527-8_3

[25] Y. Gurevich, L. Stockmeyer, and U. Vishkin. Solving NP-hard problems on graphs that are almost trees and an application to facility location problems. *J. ACM*, 31(3):459–473, 1984.

[26] G. Heiser. The Role of Virtualization in Embedded Systems. In *Proceedings of the 1st Workshop on Isolation and Integration in Embedded Systems*, pages 11–16, 2008. https://doi.org/10.1145/1435458.1435461

[27] G. Heiser, V. Uhlig, and J. LeVasseur. Are Virtual-machine Monitors Microkernels Done Right? *SIGOPS Oper. Syst. Rev. 40*, 2006. https://doi.org/10.1145/1113361.1113363

[28] M. Hohmuth, M. Peter, H. Härtig, J. S. Shapiro. Reducing TCB Size by Using Untrusted Components: Small Kernels Versus Virtual-machine Monitors. In *Proceedings of the 11th Workshop on ACM SIGOPS European Workshop*, Leuven, Belgium, pages 19–22, 2004. https://doi.org/10.1145/1133572.1133615

[29] G. J. Jacobson. Space-efficient static trees and graphs. In *Proc. FOCS*, pages 549–554. IEEE Computer Society, 1989.

[30] W. Jannen, J. Yuan, Y. Zhan, A. Akshintala, J. Esmet, Y. Jiao, A. Mittal, P. Pandey, P. Reddy, L. Walsh, M. Bender, M. Farach-Colton, R. Johnson, B. C. Kuszmaul, and D. E. Porter. BetrFS: A Right-Optimized Write-Optimized File System. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 301–315, 2015. https://doi.org/10.1145/2798729

[31] S. Joannou and R. Raman. Dynamizing succinct tree representations. In *Proc. SEA*, volume 7276 of *LNCS*, pages 224–235. Springer, 2012. https://doi.org/10.1007/978-3-642-30850-5_20

[32] S. Kannan, M. Naor, and S. Rudich. Implicit representation of graphs. *SIAM J. Discrete Math.*, 5(4):596–603, 1992.

[33] M. Lange, S. Liebergeld, A. Lackorzynski, A. Warg, and M. Peter. L4Android: A Generic Operating System Framework for Secure Smartphones. In *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, Chicago, IL, USA, pages 39–50, 2011. https://doi.org/10.1145/2046614.2046623

[34] S. Lee, and K. Park. Dynamic Rank-Select Structures with Applications to Run-Length Encoded Texts. In *Lecture Notes in Computer Science 4580*, Springer, pages 95–106, 2007. https://doi.org/10.1007/978-3-540-73437-6_12

[35] A. S. Liebergeld, M. Peter, and A. Lackorzynski. Towards Modular Security-Conscious Virtual Machines. In *Proceedings of the Twelfth Real-Time Linux Workshop*, Nairobi, Kenya, pages 25–27, 2010.

[36] J. I. Munro. Tables. In *Proc. FSTTCS*, volume 1180 of *LNCS*, pages 37–42. Springer, 1996.

[37] J. I. Munro, R. Raman, V. Raman, and S. S. Rao. Succinct representations of permutations. In *Proc. ICALP*, volume 2719 of *LNCS*, pages 345–356. Springer, 2003. https://doi.org/10.1007/3-540-45061-0_29

[38] J. I. Munro and V. Raman. Succinct representation of balanced parentheses, static trees and planar graphs. In *Proc. FOCS*, pages 118–126. IEEE Computer Society, 1997.

[39] J. I. Munro and V. Raman. Succinct representation of balanced parentheses and static trees. *SIAM J. Comput.*, 31(3):762–776, 2001. https://doi.org/10.1109/SFCS.1997.646100

[40] R. Pagh. Low redundancy in static dictionaries with constant query time. *SIAM J. Comput.*, 31(2):353–363, 2001. https://doi.org/10.1137/S0097539700369909

[41] M. Pătraşcu. Succincter. In *Proc. FOCS*, pages 305–313. IEEE Computer Society, 2008. https://doi.org/10.1109/FOCS.2008.83

[42] M. Peter, H. Schild, A. Lackorzynski, and A. Warg. Virtual Machines Jailed: Virtualization in Systems with Small Trusted Computing Bases. In *Proceedings of the 1st EuroSys Workshop on Virtualization Technology for Dependable Systems*, Nuremberg, Germany, 2009. https://doi.org/10.1145/1518684.1518688

[43] D. Peters, M. Peter, J.-P. Seifert, and F. Thiel. A Secure System Architecture for Measuring Instruments in Legal Metrology. In MDPI *Computers*, 4(2), 61 – 86, 2015. https://doi.org/10.1109/I2MTC.2015.7151517

[44] R. Raman, V. Raman, and S. S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. *ACM Transactions on Algorithms*, 3(4):Article No. 43, 2007. https://doi.org/10.1145/1290672.1290680

[45] K. Sadakane. Compressed suffix trees with full functionality. *Theory Comput. Syst*, 41(4):589–607, 2007. https://doi.org/10.1007/s00224-006-1198-x

[46] A. Velosa, J. F. Hines, H. LeHong, E. Perkins, R. M Satish. Predicts 2015: The Internet of Things. In https://www.gartner.com/doc/2952822/predicts--internet-things, 2014.

[47] J. Ziv and A. Lempel Compression of individual sequences via variable-rate coding. In *Information Theory, IEEE Transactions*, 24(5), pages 530–536, 1978.