



UNIVERSITÄT PADERBORN

Die Universität der Informationsgesellschaft

Fakultät für Elektrotechnik, Informatik und Mathematik

SmartWalk: Enhancing Social Network Security via Adaptive Random Walks

Seminar thesis

in

IT-Security

by

SIMON NACHTIGALL

submitted to:

Prof. Dr. Tibor Jäger

Paderborn, July 30, 2019

Declaration

(Translation from German)

I hereby declare that I prepared this thesis entirely on my own and have not used outside sources without declaration in the text. Any concepts or quotations applicable to these sources are clearly attributed to them. This thesis has not been submitted in the same or substantially similar version, not even in part, to any other authority for grading and has not been published elsewhere.

Original Declaration Text in German:

Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen worden ist. Alle Ausführungen, die wörtlich oder sinngemäß übernommen worden sind, sind als solche gekennzeichnet.

City, Date

Signature

Abstract

Random walks are used in many social network security systems. The random walk length has a big influence on the security and the performance of these systems. Current applications just use the fixed length random walk model, which leads to a poor trade-off between security and other desirable properties.

Thus, we look in our thesis at the new approach SmartWalk by Liu et al. [?], which uses adaptive random walks in social network security applications. SmartWalk is evaluated for three security applications Sybil defense, Anonymous communication and Link privacy and compared to the conventional fixed length model. The results of the evaluation show that the adaptive random walk model performs for all three applications better than the fixed length model.

Contents

1	Introduction	1
1.0.1	Structure of The Thesis	1
2	Motivation	3
2.1	Tree-based PPRFs	3
2.1.1	Memory consumption of PPRF-tree	4
3	Implementation	7
3.1	Requirements	7
3.2	Challenges	7
3.3	Choice of data structure	8
3.3.1	Array	8
3.3.2	Linked List	8
3.3.3	HashMap	9
3.4	Final implementation	9
3.4.1	Implementing Linked List	9
3.4.2	Implementing Punc and Eval operation	10
4	Analysis	11
4.1	Measurement Setup	11
4.2	Memory consumption	12
4.2.1	Worst-case scenario	12
4.2.2	Best-case scenario	14
4.2.3	Normal distribution scenario	15
4.3	Runtime performance	16
5	Conclusion	21
	Bibliography	23

1 Introduction

TLS (Transport Layer Security) are cryptographic protocols to provide secure communication over a computer network. If a client wants to send securely data to a server, they both have to establish a secure channel at first. Therefore in TLS 1.2 and older versions, client and server perform a 1-RTT (one round trip time) handshake protocol. Every time a client resumes a session, the 1-RTT handshake protocol has to be performed between both parties. Finally, the client is able to send securely data to the server.

The new TLS 1.3 version introduced a 0-RTT(zero round trip time) mode in order to minimize latency. When a client wants to resume a session, server and client do not need to perform a 1-RTT handshake protocol any more. The client can immediately send securely his data to the server, that is why we call it 0-RTT data. Therefore, the servers issue the client so called *session tickets*. Every time a client resumes a session, he needs to resend the issued session ticket together with its 0-RTT data. Though, the 0-RTT data sent in TLS 1.3 is not forward secure, as mentioned in the TLS specification [3]. Forward security means, that if an attacker corrupts one of the communication parties, then the security of past sessions is still guaranteed. If in TLS 1.3 the server gets corrupted, the attacker is able to compromise the 0-RTT data of past sessions.

Aviram, Gellert and Jager have shown, that you can use Puncturable Pseudorandom Functions (PPRF) to achieve forward security in TLS 1.3 0-RTT mode [1]. They present two different PPRFs for this scenario: 1. RSA-based PPRF, 2. Tree-based PPRF. In this seminar work, we want to implement the Tree-based PPRF. The size of this tree construction will grow for large servers with many clients. Thus, our implementation should mainly optimize the memory consumption. Aviram et al also gave some expectations about the consumed memory of the tree construction, which we want to confirm with our implementation results.

1.0.1 Structure of The Thesis

The goal of this thesis, is to present the PPRF-tree implementation results. Therefore, we start in Chapter 2 the Puncturable Pseudorandom Functions (PPRF) in general and then the specific PPRF-tree construction. In Chapter 3 we look at the implementation process more technically. After that, in Chapter 4 we present the main results of our implementation, which includes the memory consumption and the runtime analysis. Finally, in Chapter 5 we give a conclusion of this thesis.

2 Motivation

As said in the introduction, we want to implement an PPRF-tree construction. Therefore, we need to introduce the definition of puncturable pseudorandom functions (PPRF) at first. Then we are able to understand the main concept of an PPRF-tree construction.

PuncturablePRFS In this section, we explain the basic concept of puncturable pseudorandom functions.

A puncturable pseudorandom functions describes a special case of a normal pseudorandom function. As in a PRF, you can evaluate on a input space X pseudorandom values. Moreover, you have the option to puncture your keys. A punctured key disallows you evaluation on the inputs that have been punctured in the past. The basic definition of a puncturable pseudorandom functions is recalled from [4]:

Definition 2.1 A puncturable pseudorandom function (PPRF) with keyspace K , domain X and range Y consists of three polynomial time algorithms $\text{PPRF} = (\text{Setup}, \text{Eval}, \text{Punct})$, which are described as follows.

- $\text{PPRF.Setup}(1^\lambda)$: The Setup algorithm takes as input the security parameter λ as input and outputs a description of key $k \in K$.
- $\text{PPRF.Eval}(k, x)$: The Evaluation algorithm takes as input a key $k \in K$ and a value $x \in X$ and outputs a value $y \in Y$.
- $\text{PPRF.Punct}(k, x)$: The Puncture algorithm takes as input a key $k \in K$ and a value $x \in X$ and outputs a punctured key $y' \in K$. After puncturing, evaluation on x is disallowed ($\text{PPRF.Eval}(k', x = \emptyset)$) is disallowed.

The security analysis of the PPRFs is not necessary for our PPRF-tree implementation, hence we will skip it.

2.1 Tree-based PPRFs

The tree-based PPRF construction is based on the GGM (Goldreich-Goldwasser-Micali) [2] construction. Hence, we start with the explanation of the tree-based GGM construction, then we will describe the transformation to a PPRF.

Let $G : \{0, 1\}^\lambda \rightarrow \{0, 1\}^{2\lambda}$ be a pseudo random generator (PRG). We assume that k is random seed value. Then we define $G_0(k), G_1(k)$ as the first and second

half of string $G(k)$. A GGM construction is defined as a binary tree on the input domain of our PRF, where every leaf node represents an evaluation of the PRF. Every edge to a left child is labeled with 0 and every edge to a right child is labeled with 1. Then, we can label every node $x = x_1 \dots x_n \in \{0, 1\}^n$ depending on the edge values x_i on the path from the root node to x . In order to evaluate on the PRF input x , we compose G along the path from root node to leaf node x and output $(G_{x_n} \circ \dots \circ G_0)(k) \in \{0, 1\}^\lambda$. Figure 4 illustrates an evaluation of leaf node $x = 011$ with random seed k .

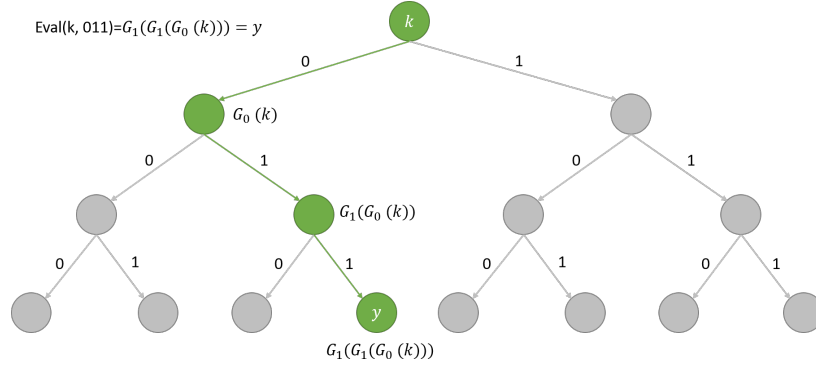


Figure 2.1: This figure illustrates the structure of the implemented linked list. The size of one element is 36 Byte.

The GGM-tree based construction for PRFs can be transformed to puncturable PRFs [?], which works as follows. We define x_{child} as a child of path $p = (node_1, \dots, node_k)$, if x_{child} lies not on path p and x_{child} has a parent node in path p . If we want to puncture at input $x = x_1 \dots x_n \in \{0, 1\}^n$, we need to calculate the evaluation of all childs of path $p_{eval} = (x_1, x_1x_2, \dots, x_1 \dots x_n)$. The output k' contains the evaluations of all child nodes $x_1, x_1\bar{x}_2, \dots, x_1x_2 \dots \bar{x}_n$. Then, we can discard our random seed k . After that, all puncturing and evaluation operation are performed on the updated key k' , but evaluation on input x is disallowed. The puncture operation in figure ?? leads to an updated key k' , which contains three evaluated nodes. The nodes in the updated key k' represent the root nodes of the remaining binary subtrees.

2.1.1 Memory consumption of PPRF-tree

We need to store in our key k' a certain number of evaluated nodes from the binary tree. The number of nodes depends on, where we puncture in our binary tree. In the following, we present three different puncturing scenarios and their influence on the key size. Especially, we want to know for each scenario the peak size of our key.

Best-Case Scenario In the best case scenario, we start puncturing with the most left leaf node $x = 0$ and then puncture in ascending order the leaf nodes

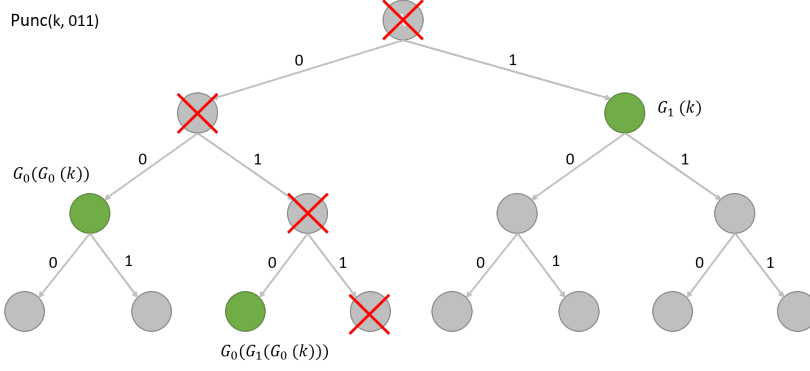


Figure 2.2: This figure illustrates the structure of the implemented linked list. The size of one element is 36 Byte.

from left to right. After puncturing the first node, we reach the peak size of our key with $peak_{best} = depth(PPRF\text{-}Tree) = O \log n$ nodes (n number of nodes in PPRF-Tree). After puncturing the first half of the nodes, our key contains only the root node of the right subtree. After puncturing all possible inputs, our key is empty. There is no other scenario with lower peak size of key k .

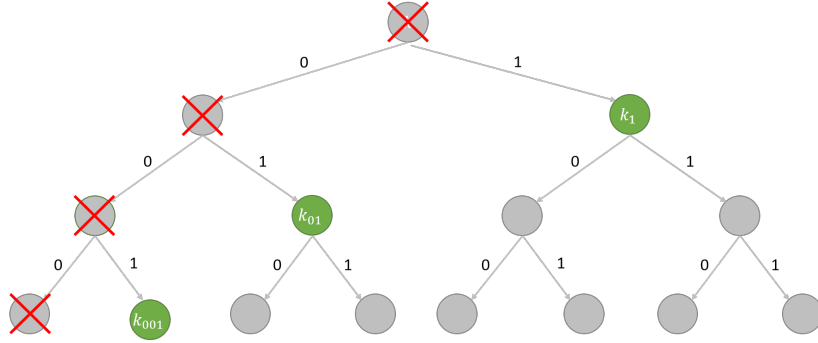


Figure 2.3: This figure illustrates the structure of the implemented linked list. The size of one element is 36 Byte.

Worst-Case Scenario In the worst case scenario, we start puncturing with the most left leaf node $x = 0$ and puncture in ascending order every second leaf node from left to right. In this case, our key contains all leaf nodes that have not been punctured yet, which is every second leaf node in the binary tree. That means, we have to store $peak_{worst} = \frac{n}{4} = O(n)$ nodes. Before reaching the peak size, the key size increases monotonically after every puncture operations. There is no other scenario with higher peak size of key k .

Normal distribution Scenario Aviram et al. presented a scenario, where leaf nodes in a certain window are punctured [1] with a higher probability. Therefore,

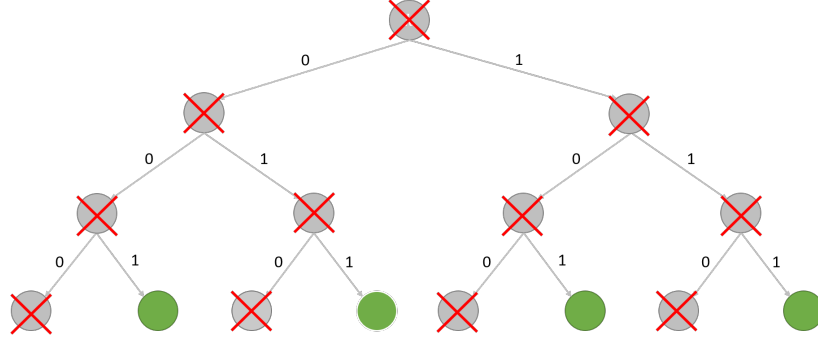


Figure 2.4: This figure illustrates the structure of the implemented linked list.
The size of one element is 36 Byte.

we generate random values depending on our normal distribution N with standard deviation σ , which illustrates the window size. In this thesis, we want to find out, how the window size influences the key size.

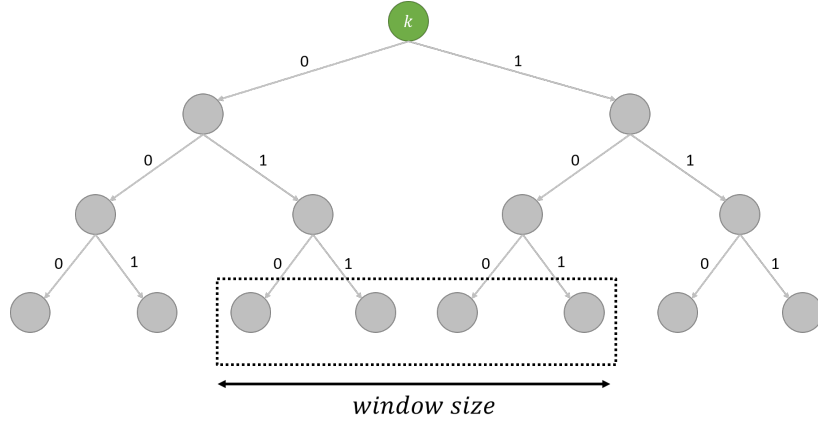


Figure 2.5: This figure illustrates the structure of the implemented linked list.
The size of one element is 36 Byte.

3 Implementation

In this chapter, we want to look at my implementation of the presented PPRF-tree construction. First of all, we need to state the requirements of the implementation.

3.1 Requirements

The implementation shall consist two main components:

1. Implement the presented PPRF operations *setup*, *eval* und *punc*.
2. Storage of the remaining subtree root nodes in a compatible data structure.

The data structure can get for greater tree depths very large, therefore we want to optimize the memory consumption of the data structure. That is the reason, why we have to find a memory efficient data structure.. Our main focus is optimizing the memory consumption, though the run time should be acceptable. Additionally, we want to confirm the memory estimations from the presented worst and best case scenario (ZITAT).

Furthermore, we need to consider some technical requirements for our implementation. The programming language is C, because later this tree construction shall be integrated into OpenSSL, which is written in C. In C, you can do memory mangement completly on your own. Thus, you can implement with less memory overhead in comparison to Object-Orientated Programming languages like Java or C++.

3.2 Challenges

Before implementing the PPRF-tree, we need to point out the challenges for this purpose.

- Often, there is a tradeoff between memory consumption and run time. If we want to optimize the memory, we need to be careful that our run time will not grow excessevely.
- Moreover, the memory consumption of a C program is influenced by the operating system and the memory hardware. We have to be careful, that we do not optimize the memory only for our testing machines.

- Another challenge is, that our implementation is part of an security protocol. That means, we store secret data in our implementation and it shall be impossible to reconstruct this data after deletion.

3.3 Choice of data structure

As mentioned before, we have to select a memory efficient data structure for our implementation. Therefore we look at ad- and disadvantages of three standard data structures.

3.3.1 Array

Advantages

- An array is very memory efficient, if you need to store a fixed number of elements.
- In a sorted array, you can search elements very fast in $O(\log n)$ (n : number of elements)

Disadvantages

- The array size is fixed. Though, our implementation requires a flexible size. This is also possible in an array by expensive allocation/deallocation of memory. Altogether this leads to a lot of memory and esapcially management overhead.

3.3.2 Linked List

Advantages

- A Linked List is a dynamic data structure.
- Implementation is very easy.

Disadvantages

- Each element in the linked list needs some small additional memory overhead.
- Search operation in a linked list can only performed in $O(n)$ (n : number of elements)

3.3.3 HashMap

Advantages

- Operations as add/delete/search can be performed quickly in $O(1)$

Disadvantages

- The size of the hash map is fixed. Similar to the array data structure, adapting the size leads to memory and management overhead.
- To avoid hash collisions, the hash maps has empty buckets by a factor of two, which leads to a big memory overhead.

The array and hash map are espacially good for run time optimizations. Linked list is a good data structure for less memory overhead, if your amount of data is flexible. Because we want to optimize memory consumption and our data size is flexible, we decide for implementing the linked list at first.

3.4 Final implementation

S

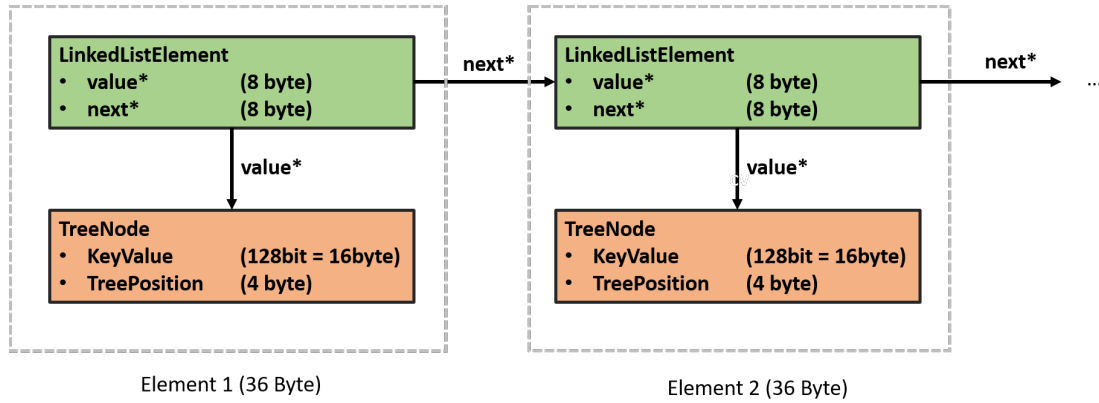


Figure 3.1: This figure illustrates the structure of the implemented linked list. The size of one element is 36 Byte.

3.4.1 Implementing Punc and Eval operation

Every Punc/Eval on input x starts with a search operation in the linked list. We have to find the responsible subtree root node, which contains the leaf node x .

The Punc/Eval operation need a PRG in order to evaluate new nodes in the binary tree. As suggested in [1], we use the hash function SHA-3 implementation. Because our security parameter λ is fixed to 128 bit, we choose 256 as the output

size of our hash function. For left child evaluation of node k we cut the first 128 bit of $H_{sha3}(k)$, for the right child we cut the last 128 bit of the output.

3.4.2 Implementing Linked List

Figure 3.1 shows the final structure of our linked list implementation. One element in the linked list consists two parts:

1. The structure for linked list overhead (green box): Here, we need to store 2 memory addresses of 8 byte: One to the next element, and one to the containing value of the linked list.
2. The tree node structure, which is the value of the linked list element (orange box). The tree node represents one subtree root node in our PPRF-tree. We have store the key value (128 bit = 16 byte) and the node position (4byte) in the PPRF-tree.

In total, the size of one element in the linked list is 36 byte. We have some additional constant memory overhead for our linked list in total, which is 24 byte. Forumlar 3.1 shows the expected memory size of our linked list.

$$size_{list} = 36n + 24 \quad [\text{Byte}] \quad (3.1)$$

where:

n : = number of elements in linked list
 $size_{list}$: = expected size of linked list in Byte.

4 Analysis

In this section, we analyze the performance of our implementation. First, we will describe our measurement setup and tools. Second, we evaluate the memory consumption of our implementation. Finally, we analyze the runtime performance of different operations in the PPRF Tree.

4.1 Measurement Setup

Hardware Setup We execute all performance test on a Ubuntu 64-bit virtual machine. Our machine uses an Intel Core i5-7200 CPU @ 2.50 GHz with 5 GB RAM

Tools In order to measure our performance, we need different tools. The secret key is dynamically allocated on the memory heap. Therefore, we need to measure the memory heap size of our implementation. We use *Valgrind-Massif* as a measurement tool for the memory heap size. It distinguishes between the effective memory heap size and the total memory heap size. The effective memory heap size contains the allocated heap memory of the programmer, the total memory heap size contains additionally memory overhead (e.g padding for performance). For runtime measurements in our implementation, we use the *time* C-library.

Simulation Setup A simulation run gets three inputs:

1. *security parameter* λ : This parameter is fixed for all simulations to 128 bit.
2. *tree depth* d : This parameter states the depth of the PPRF-tree.
3. *scenario*: As explained in TODO, there are different scenarios how we puncture our PPRF tree. In order to show, that implementation works correctly we simulate each scenario and evaluate, if the memory behaviour confirms our assumptions.
 - *Best case scenario*: In this scenario, we puncture our leaf nodes from left to right. We start with leaf node 1 and then puncture every leaf node ascending.
 - *Worst case scenario*: In this scenario, we puncture our leaf nodes from left to right. We start with leaf node 1 and then puncture every second leaf node.

- *Normal distribution scenario:* In the last scenario, we puncture leaf nodes depending on the normal distribution. The standard deviation σ of the normal distribution is another parameter for the simulation of this scenario. The mean of the standard deviation is fixed half of leaf nodes size. In every simulation step, we generate a normal distributed leaf number x . This leaf node x is punctured in the PPRF tree

4.2 Memory consumption

We begin the evaluation with the observed memory consumption of the PPRF tree in our simulation. Therefore, we analyze the 3 presented scenarios.

4.2.1 Worst-case scenario

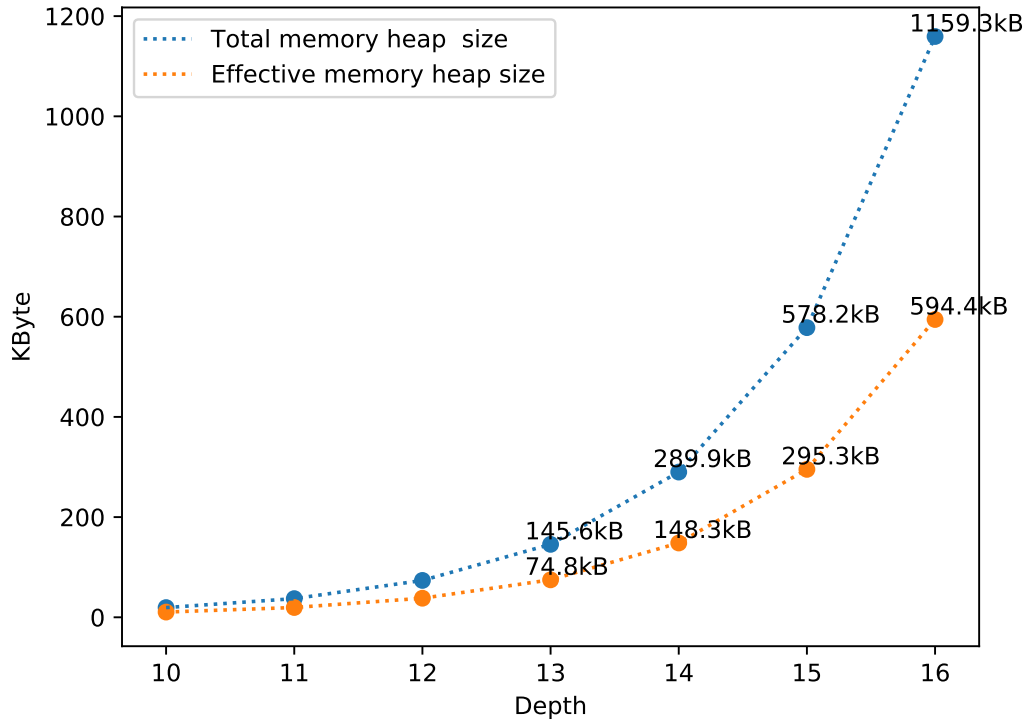


Figure 4.1: Memory consumption of our implementation depending on the depth of our PPRF-tree. The orange line shows the measured effective memory heap size and the blue line shows the total memory heap size.

Figure 4.1 shows the measured effective and total memory heap size in KByte for different PPRF-tree depths. First, we focus on the effective memory heap size (orange points in plot): As we can see, the results confirm our estimations from formular 3.1. If we calculate for example for tree depth 15 our expected memory size (295 KByte), we get very close to the measured value 295.3 Kbyte. Furthermore, the measured memory size grows exponentially. This confirms our expectations again, because the leaf nodes double by increasing the tree depth by 1. Therefore, in the worst case scenario we have to store twice as much leaf nodes in our data structure.

Additionally, Figure 4.1 shows the total memory heap size, which includes added memory overhead. The total memory heap size is nearly twice as big as the effective memory, which means half of the total memory is overhead. In order to reduce it, we have to find out, how the overhead is combined.

- One reason for overhead is that structure members are padded to 'natural' address boundaries for performance reasons. This depends heavily on the address size of your system (normally 32/64 bit) On the test machine (64-bit address size) the Tree node structure has an effective memory size of 20 byte, though including padding 32 byte.
- Another reason is, that there is a certain amount of memory overhead (usually 8 byte) associated with every allocation on the heap. Our implementation has two memory allocations for one element in the linked list, one for the linked list structure and one for the tree node structure. That means, for every element in the linked list we have 16 bytes additonal overhead. Altogether, the effective memory size is 36 byte and the total size is 64 byte.

One idea to reduce the overhead is to use only one structure in total per element. As a result, we do not need to store a pointer to the linked list content and we save 8 byte for the memory allocation. The new structure 4.2 has an effective size of 28 byte, and would be padded to 32 byte. With the overhead for the memory allocation, the total memory size for one element would be 40 byte.

TreeNode	
• next*	(8 byte)
• KeyValue	(128bit = 16byte)
• TreePosition	(4 byte)

Figure 4.2: This figure shows the new proposed structure for a linked list element. The total size including of one element is 40 Byte.

Furthermore, we want to know, how the memory consumption behaves in one simulation. Figure 4.3 illustrates, that the memory size grows monotonically. In the worst case scenario every puncture operation leads to more nodes to store, so over time the data structure grows as we can see in the measurement.

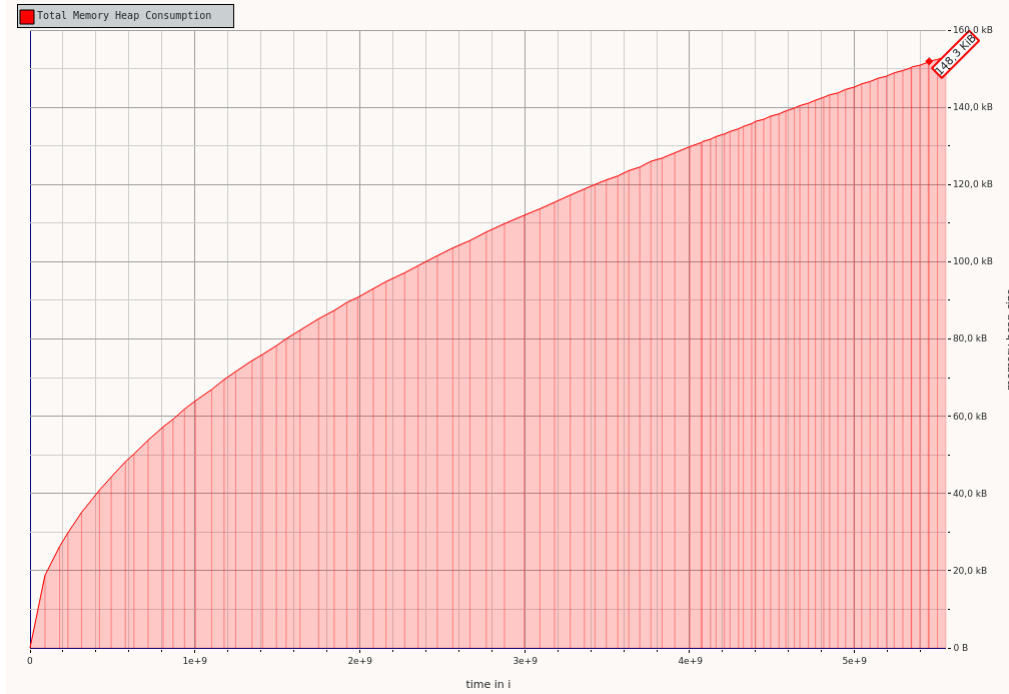


Figure 4.3: The memory behaviour in a single simulation with PPRF-tree depth 14 in the worst case scenario.

4.2.2 Best-case scenario

Figure 4.4 shows the measured total memory heap size in KByte for different PPRF-tree depths. We refrain from analyzing the memory overhead in this scenario, because this would be similar to the worst case scenario. The total memory size only grows linearly, in contrast to the worst-case scenario. As explained in TODO, our maximum number of lead nodes, we need to store, are the depth of the PPRF tree. By increasing the tree depth by one, the data structure additionally contains one more element at peak. The measured memory size is as expected very low. For all depths, we are still under 3 Kbyte, which is in practice ignorable in comparison to the worst case scenario.

If we look at the memory behaviour over time in a simulation with depth 14 (Figure 4.5), we can see that the peak point of the memory consumption is right after the beginning. In the best case scenario, we have store after the first puncture

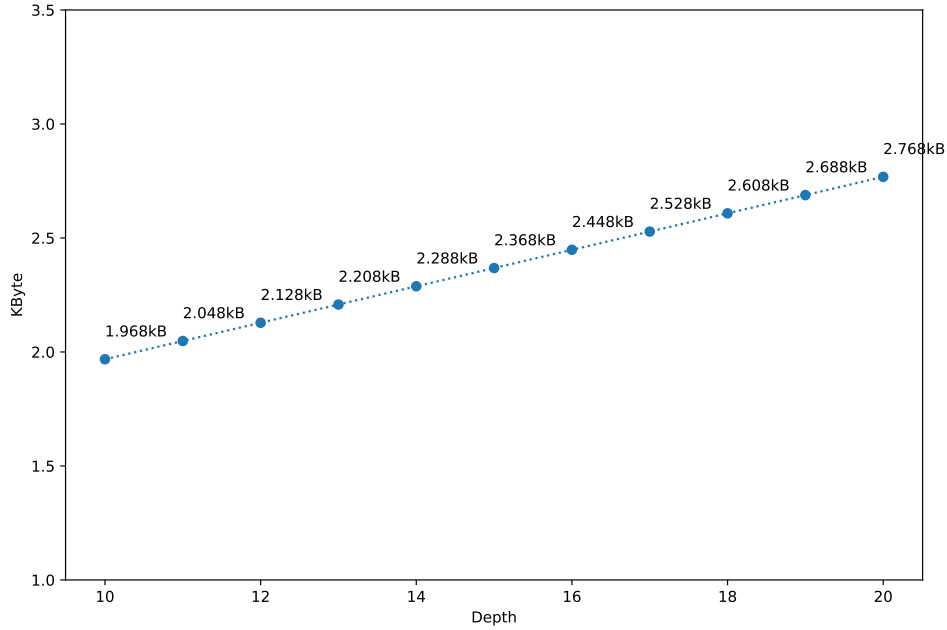


Figure 4.4: The figure shows the total memory consumption of our implemented PPRF-tree construction in the best case scenario depending on PPRF-tree depth

operation depth of the tree nodes. That is the maximum number in the whole simulation.

4.2.3 Normal distribution scenario

In the normal distribution scenario, we want to find out how the standard deviation influences the memory consumption. Therefore, we look at two example simulations of tree depth 14. Figure 4.6 shows the memory heap size for $\sigma_1 = \frac{\#leaf\ nodes}{5} = 2048$ and 4.7 for $\sigma_2 = \frac{\#leaf\ nodes}{20} = 512$. as we can see, the peak heap size for σ_1 is 104 KByte and for σ_2 32 KByte. A four times smaller standard deviation leads nearly to three times smaller peak size. The intuition is that with a greater σ , we puncture more elements far away from the middle leaf nodes. This leads to a greater number of subtree, we need to store. A smaller σ leads to more punctures in the middle leafs node area. In this case the possibility is higher, to delete nodes in the subtree, because the whole subtree has been punctured.

Another interesting observation is the memory behaviour in both cases. The memory peak is reached right after the beginning of the simulation and then the memory size decreases. In the case with smaller σ the memory size converges faster.

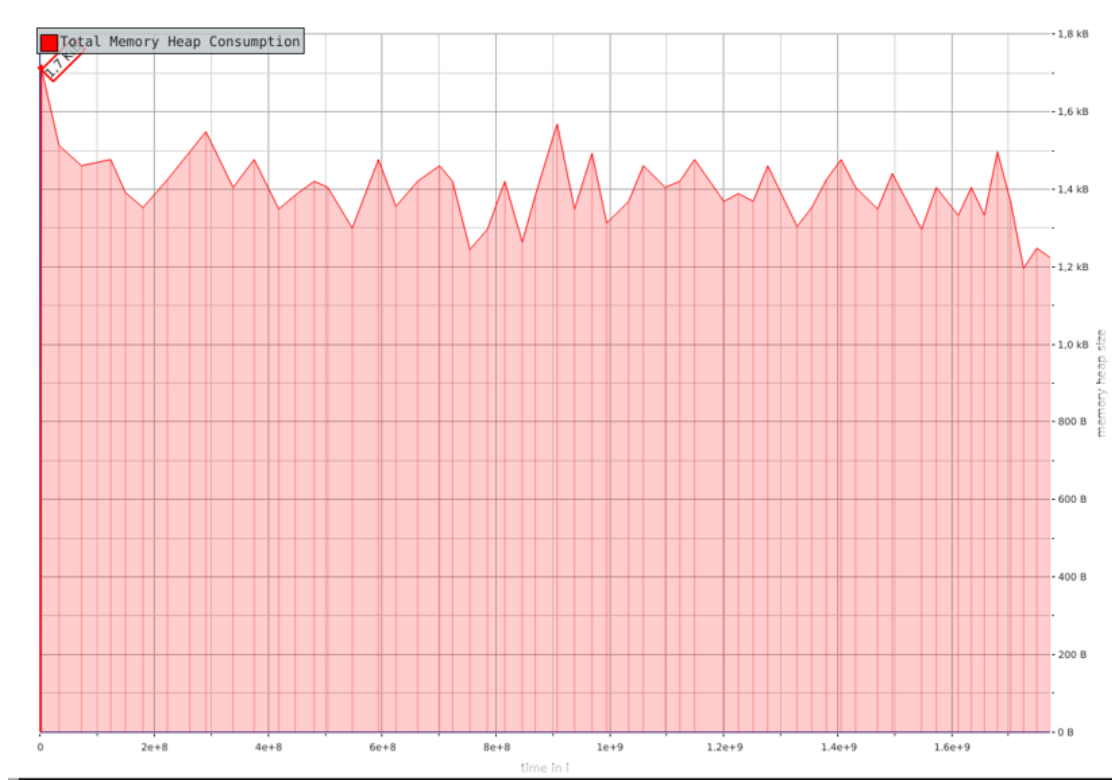


Figure 4.5: The memory behaviour in a single simulation with PPRF-tree depth 14 in the best case scenario.

4.3 Runtime performance

In this section, we want to evaluate the runtime performance of the puncture and evaluation operation. We want to know how the data structure size and the number of hash evaluations influence our runtime.

We do analysis only in the worst-case scenario, because in the best case scenario the data structure is too short to notice significant differences in runtime. First, we want to focus on the puncture operation.

Figure ?? shows one example simulation of the worst case scenario with tree depth 14. One data point (x, y) illustrates the execution time y of the puncture operation at time x . The execution time grows linearly over time until about one ms in the end. As we have seen in figure 4.1, the data structure size grows over time. It seems that the greater data structure causes a higher execution time for the puncture operation. We can confirm this observation, if we look at the mean puncture execution time for different tree depths (blue points) in figure ??. If we increase the tree depth by one, the mean execution time nearly doubles. Altogether, the execution time grows exponentially just as the total memory heap

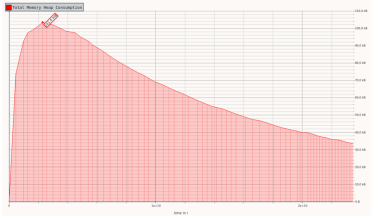


Figure 4.6:

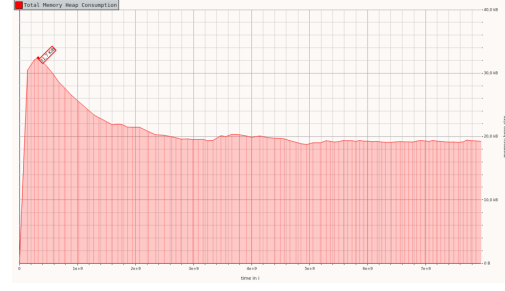


Figure 4.7:

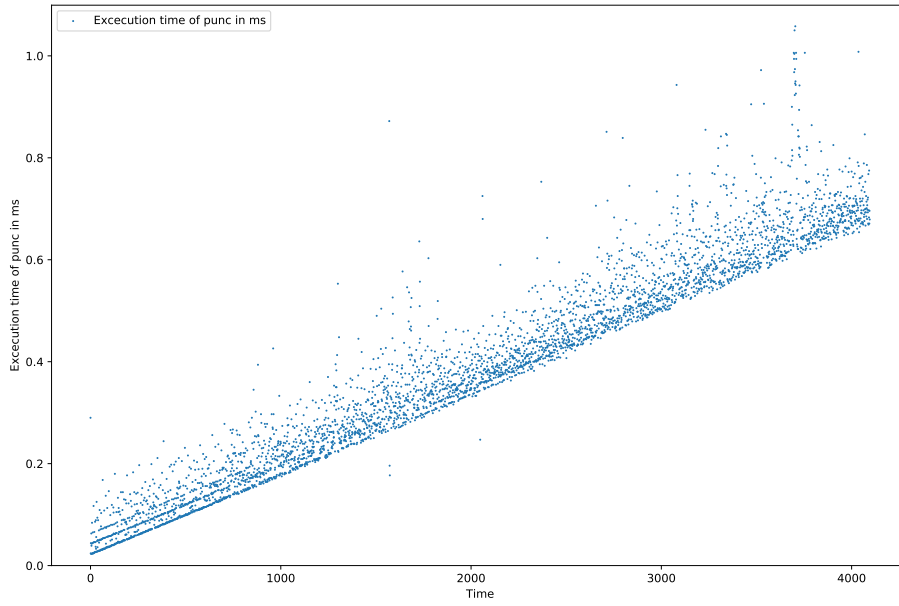


Figure 4.8: Figure shows a simulation with PPRF-tree 14 in the worst case scenario. The blue points display the execution time of single puncture operations in the simulation

size in figure 4.1. That means, that there is a strong correlation between the data structure size and the execution for the puncture operation.

In order to find out the exact reason for the increasing execution time, we additionally need to consider the time for hash evaluation in the puncture operation. The orange points in figure 4.9 show the execution time for a puncture operation without the hash evaluation. The difference between the blue and the orange points illustrate the execution time for the hash evaluations. If the depth is increased, the hash evaluation time stays nearly constant. Therefore, we know

that the data structure size causes the increasing puncture time.

If we look more closely to the procedure of the puncture operation (TODO), we notice that at the beginning the responsible subtree node for the entered counter x has to be searched in the linked list. This search operation takes $O(n)$ time, where n is the number of elements in the linked list. If we increase the number of nodes in our data structure, the search operation will take longer and lead to higher execution time.

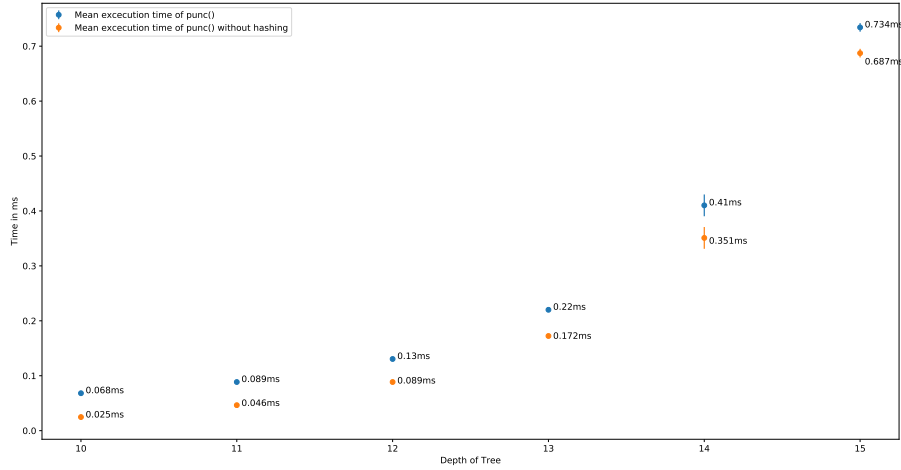


Figure 4.9: This figure shows the mean execution time of the puncture operation depending on the PPRF-tree depth. Furthermore, all data points contain error bars. The blue points illustrate the mean execution time for the puncture operation. The orange points show the mean execution time for the puncture operation without performing the hash evaluation operations

So far, we analyzed the runtime for the puncture operation, but we are missing the evaluation for the evaluation operation. Figure 4.10 shows the execution time of single evaluation operations (orange points) in comparison to puncture operations (blue points). The execution time for evaluation behaves nearly the same as the puncture operation. This is because at the beginning of the evaluation operation, the correct subtree has to be searched in the linked list, too. Though, the execution time is slightly smaller, because we do not perform any modifying operations on the linked list.

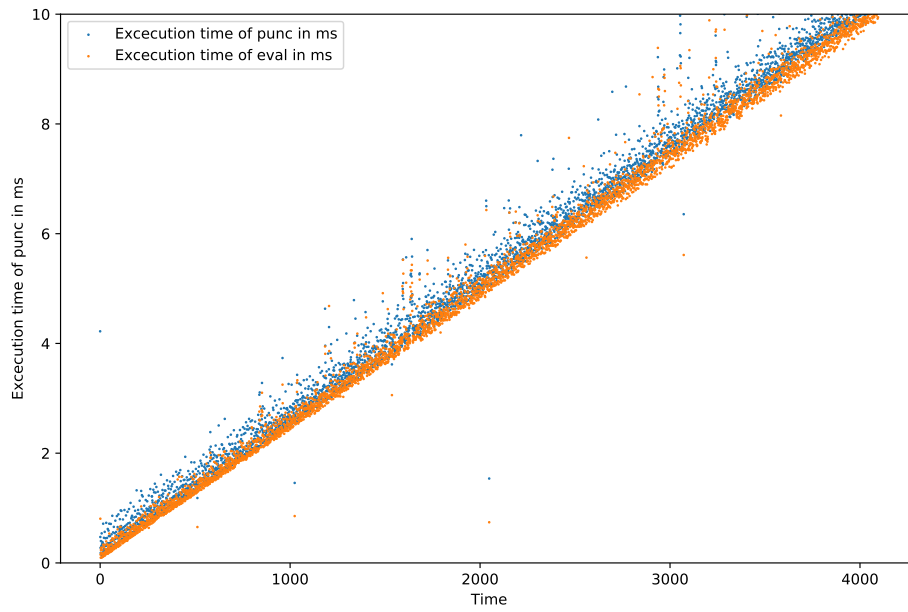


Figure 4.10: Figure shows the execution time of single puncture operations (blue points) and evaluation operations (orange points) in a simulation over time . The PPRF tree depth of this simulation is 14 and the simulated scenario is the worst case scenario.

5 Conclusion

In this thesis, we evaluated the implementation results of our PPRF tree implementation. We simulated our PPRF-free for three different scenarios. We could confirm the memory consumption assumptions from the paper by Aviram et al [1]. Moreover, we noticed a lot of additional memory overhead and we presented a solution to reduce the overhead significantly.

In the third simulation scenario we used the normal distribution for puncturing leaf nodes in the PPRF-tree. The presented simulation consist only a few example simulation runs. In the future, we need to do a more detailed analysis to get a more valuable results.

Moreover, we evaluated the runtime performance of our implementation. We noticed a significant increase of runtime performance for large sizes of our data structure. The reason behind this observation is, that the search operation in a linked list needs $O(n)$ (n : number of elements) time steps. An alternative data structure would be a binary search tree, which can perform a search operation in $O(\log n)$. Furthermore it needs no additional memory in comparison to the linked list. In the future, the binary search tree for our PPRF-tree construction shall be implemented and evaluated.

Bibliography

- [1] Nimrod Aviram, Kai Gellert, and Tibor Jager. Session resumption protocols and efficient forward security for tls 1.3 0-rtt. Cryptology ePrint Archive, Report 2019/228, 2019. <https://eprint.iacr.org/2019/228>.
- [2] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. How to construct random functions. *J. ACM*, 33(4):792–807, August 1986.
- [3] Eric Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446, August 2018.
- [4] Amit Sahai and Brent Waters. How to use indistinguishability obfuscation: Deniable encryption, and more. In *Proceedings of the Forty-sixth Annual ACM Symposium on Theory of Computing*, STOC '14, pages 475–484, New York, NY, USA, 2014. ACM.