Master Thesis

Julius-Maximilians-
UNIVERSITÄT
WÜRZBURG

Evaluating the privacy of contact discovery: Hash reversal and TEEs

Christoph Sendner
Department of Computer Science
Chair of Computer Science II (Secure Software Systems)

Prof. Dr.-Ing. Alexandra Dmitrienko
First Reviewer

Christoph Hagen
First Advisor

Submission
20. July 2020

www.uni-wuerzburg.de

Abstract

Currently deployed contact discovery of mobile messengers is based on the transmission of phone numbers to the service provider. This information is private and anonymized by hashing them. In this work, we show, that this anonymization is pseudo-anonymous and can easily be broken by an attacker.

For that, we develop two hash reversal techniques: one using brute-force approach and another one using look-up databases. We provide generic architectures for each of the approaches. Additionally, we provide and compare two instantiations for each. Furthermore, we evaluate and compare them to the third approach based on rainbow tables.

The evaluation shows near instant lookup-times of under 0.1 ms using in-memory lookup databases, this approach is however costly in terms of memory – it would require over 10 TB RAM, which would be difficult to obtain. Our brute-force approach shows an astonishing performance, being able to reverse any mobile number in under 100 seconds using consumer-level hardware. The rainbow tables produce lookup-times of 4.5 minutes with a success rate of over 99.99%.

The results of our evaluation demonstrate, that hash reversals of mobile phone numbers are practical and near instant. Thus, an attacker can easily reverse hash digests of mobile phone numbers and de-anonymize personally identifiable information – like phone numbers transmitted to the service provider of mobile messenger apps.

Zusammenfassung

Die derzeit angewandte Contact Discovery Methode in Mobile Messengern basiert auf einer Übertragung aller Telefonnummern des Nutzers an den jeweiligen Anbieter. Die Telefonnummern gelten als privat und werden durch Hashing anonymisiert. In der vorliegenden Arbeit zeigen wir, dass diese Anonymisierung nur pseudo-anonym ist und von einem Angreifer leicht umgekehrt werden kann.

Zu diesem Zweck entwickeln wir zwei Hash-Umkehrtechniken: Eine verwendet Brute-Force und die andere Datenbanken. Wir stellen generische Architekturen für beide Herangehensweisen bereit. Zusätzlich erstellen und vergleichen wir jeweils zwei Instanziierungen der besagten Architekturen. Darüber hinaus evaluieren und vergleichen wir beide Ansätze mit einem dritten Ansatz basierend auf Rainbow Tables.

Die Evaluierung zeigt ein nahezu sofortiges Brechen von Hashes innerhalb von unter 0,1 ms unter Verwendung von In-Memory-Datenbanken. Dieser Ansatz geht jedoch sehr zulasten des Arbeitsspeichers – eine Datenbank würde über 10 TB RAM erfordern, was schwierig zu bewerkstelligen ist. Unser Brute-Force-Ansatz jedoch zeigt eine erstaunliche Leistungsfähigkeit, da dieser jede Mobilfunknummer mithilfe von handelsüblicher Hardware in weniger als 100 Sekunden wiederherstellen kann. Die Rainbow Tables ergeben Suchzeiten von 4,5 Minuten mit einer Erfolgsrate von über 99,99%.

Die Ergebnisse unserer Evaluierung zeigen, dass Hash-Umkehrungen von Mobiltelefonnummern praktisch sind und nahezu sofort verlaufen. Auf diese Weise kann ein Angreifer leicht Hash-Digests von Mobiltelefonnummern rückgängig machen und persönlich identifizierbare Informationen de-anonymisieren – wie zum Beispiel Telefonnummern, die an den Anbieter einer Mobile Messenger-App übertragen werden.

Contents

1. Introduction **1**

2. Background **5**
2.1. Phone Numbers . . . 5
2.2. Hashes . . . 6
2.3. Redis . . . 7
2.4. LMDB . . . 9
2.5. Rainbow Tables . . . 13
2.6. Rainbow Crack . . . 15

3. Problem Statement **17**
3.1. Telegram . . . 17
3.2. WhatsApp . . . 18
3.3. Snapchat . . . 19
3.4. Threema . . . 19
3.5. WeChat . . . 20
3.6. Viber . . . 20
3.7. Wire . . . 20
3.8. Signal . . . 21
3.9. Attacker Model . . . 22
3.10. Discussion . . . 22

4. Related Work **25**
4.1. Private Set Intersection . . . 25
4.2. Enumeration Attacks . . . 26
4.3. Hash Reversal . . . 27

5. Brute-force Hashes **29**
5.1. Architecture . . . 29
5.2. Instantiation of Architecture using Hashcat . . . 32
5.3. Instantiation of Architecture using JTR . . . 35

6. Hash Database **37**
6.1. Architecture . . . 37
6.1.1. Components . . . 37
6.2. Instantiation with Redis . . . 39
6.2.1. Implementation of Look-Up Table Generation Unit . . . 39
6.2.2. Implementation of Phone Number Database . . . 41
6.2.3. Implementation of Hash Reversal Unit . . . 43
6.3. Instantiation with LMDB . . . 43
6.3.1. Implementation of Look-Up Table Generation Unit . . . 44
6.3.2. Implementation of Phone Number Database . . . 45
6.3.3. Implementation of Hash Reversal Unit . . . 46

7. Evaluation **49**
7.1. Test Setups . . . 49
7.1.1. Test Setup – OpenStack . . . 49
7.1.2. Test Setup – SLURM . . . 49
7.1.3. Test Setup – 1080TI . . . 50
7.1.4. Test Setup – Fujitsu . . . 50
7.1.5. Test Setup – Manjaro VM . . . 50
7.2. Evaluation of Brute-Force . . . 51
7.2.1. Comparison JTR and Hashcat . . . 51
7.2.2. Reversing Entire Mobile Phone Number Space with Hashcat . . . 52
7.2.3. Reversing Entire Mobile Phone Number Space with JTR . . . 55
7.2.4. Discussion . . . 57
7.3. Evaluation of Database . . . 59
7.3.1. Evaluation of Redis . . . 59
7.3.1.1. Test 1 with Redis . . . 60
7.3.1.2. Test 2 with Redis . . . 61
7.3.2. Evaluation of LMDB . . . 62
7.3.2.1. Tests with LMDB . . . 65
7.3.3. Comparison of Databases . . . 69
7.4. Evaluation of Rainbow Crack . . . 73
7.4.1. Evaluation Chain Length and Count . . . 73
7.4.2. Evaluation Reversing All Mobile Phone Number . . . 75
7.4.3. Evaluation of Size/Performance Trade-off . . . 78
7.5. Discussion . . . 79

8. Conclusion and Future Work **83**

Bibliography **85**

Appendix **95**
A. Config for Cluster-Creation Redis . . . 95
B. B+-Tree Data of LMDB Evaluation . . . 96

1. Introduction

Mobile phones have become a very common technology nowadays. Their usage and spread is on a rise since the first cellphones were built. Their utilization became ubiquitous with the inception of smartphones, that allowed users to not just make a call, but opened the door for a wide range of various applications.

One of those new applications are so-called messenger apps – such as WhatsApp [1]. First adapted as a more cost-effective alternative to SMS, they became the de-facto way of commutation on smartphones. They extended their initial text-based channels with voice and video calls.

Messenger apps require contact information about other users, such that it is possible to make calls or send messages. Contact lists of users might be pretty large and difficult to establish manually, hence the apps provide a mechanism to simplify contact bootstrapping through the so-called Contact Discovery method. In this method, users allow an app to get access to their contact lists established by other apps, for instance, to e-mails stored by an e-mail client or to the contact list with phone numbers.

In this work, we concentrate on mobile messenger apps, that use user's mobile phone numbers as contact information. In such apps, the users share their contact lists in form of phone books with the service provider. Those contacts are then intersected with all users registered for the same service, and the overlap is added to the user as newly discovered contacts. However, this approach has two inherent privacy problems: Possibility of enumeration attacks and leaking personally identifiable information.

Enumeration attacks. Enumeration is a method, where a malicious external user can abuse the Contact Discovery process to get all registered users. In the case of phone books, a malicious user can just define a fake phone book with mobile numbers, query the provider and get a list of registered users back. By repeating this process, an attacker could query all possible phone numbers and by that get all registered users. This attack vector was explored in depth by the authors of [2].

Leaking private information. In the Contact Discovery process, entire contact lists of phone numbers are shared with the provider. This includes already registered users as well as not registered users. Since a phone number is defined as an identifiable information under the General Data Protection Regulation (GDPR) [3] and non-registered users didn't give their consent to share their phone numbers with the provider, their numbers need to be anonymized.

Therefore providers, such as WhatsApp or Signal [4], hash all phone numbers of a contact list before transmitting them to their servers. Hash functions are used, because of their one-way property – once a number is hashed, there is no easy way to reverse the hash, or digest, back to the phone number. The only way to reverse a secure hash is brute-forcing. This attack is supposed to be difficult, because the search space of phone numbers spans trillions of numbers. One would need to calculate the hash of every number to reverse a digest with certainty.

However, we're still operating within the predictions of Moore's law [5] and increase our computational power rapidly. What seemed infeasible to compute in the past, becomes within reach of today's computers. So, it might be feasible for modern attackers to crack hashes and reveal identifiable information about users.

The problem of leaking private information through the Contact Discovery was pointed out in previous research [6], where authors suggested to apply Private Set Intersection (PSI) methods [7] to make it possible to intersect the contact lists of the server provider and of the user in a privacy-preserving manner without the need to transfer entire user contact lists to provider's servers. While PSI protocols generally solve the problem of leaking personal identifiable information in the Contact Discovery process, they currently fail to scale with the demand and user-base of mobile messengers.

In this thesis, we aim to explore the problem of leaking private information in Contact Discovery process in mobile messengers. In particular, we analyze Contact Discovery methods used by mobile apps such as WhatsApp, Signal and others and evaluate several techniques for hash reversal. In particular, we develop two new methods: Hash reversal through brute-forcing and building a lookup database, and compare these two methods and one additional method developed in [2] based on rainbow tables to evaluate their performance and feasibility to efficiently revert hashes. Results of this thesis show, that difficulty of hash reversal of hashes computed from mobile phone numbers is low and any phone number can be reversed as fast as in less than 100 seconds using consumer-grade hardware. This implies that the difficulty of hash reversal of mobile phone numbers is largely overestimated and current anonymization methods are almost as good as no protection.

Our contributions. In particular, we make the following contributions:

(a) We analyze the following mobile messenger apps: Telegram [8], WhatsApp, Snapchat [9], Threema [10], WeChat [11], Viber [12], Wire [13] and Signal. We find out that neither Telegram, Snapchat, Wechat nor Viber apply any anonymization to the contact discovery process. Messenger apps like WhatsApp, Threema, Wire and Signal, offer anonymization through hashing of the phone number using a SHA-algorithm.

(b) We explore hash reversal using brute-force technique. For that, we propose an architecture and instantiate it using the cracking tools Hashcat [14] and John the Ripper (JTR) [15]. We show that JTR outperforms Hashcat in execution time by nearly a 85-fold performance gain and can achieve near ideal hash rate possible on our consumer-grade hardware with JTR.

(c) We explore hash reversal using a lookup database. Here, we create the database that includes key-value pairs for mobile phone numbers and their respective hashes. We developed a generic architecture and instantiated it with two database technologies: Redis [16] and Lightning Memory-Mapped Database (LMDB) [17]. Our comparison of both instantiations shows that LMDB outperforms Redis in terms of memory consumption and lookup-times. While this technique is memory-consuming, it allows us to achieve near instant lookup times.

(d) We evaluate and compare three hash reversal techniques: Brute-forcing and lookup database – both developed in this thesis – and another technique [18] that is based on rainbow tables [19]. Our results show that brute-forcing shows the most promising results, with traversing the entire space in 93.2 seconds, while searching for 10^6 digests using consumer-grade hardware. The database approach requires more than 10 TB of memory to hold the entire database of mobile phone numbers and their digests – something which could be feasible for state-level attacker and service provider, but it provides near instant lookup-times of 5.5 ms with our architecture. Further, we achieve lookup-times of 4.4 minutes with a success-rate of over 99.99% using rainbow tables.

Outline. We first describe in Chapter 2 the background on phone numbers, hashes, Rainbow Tables and utilized tools. Next, in Chapter 3, we state the problem, compare different mobile messenger apps and outline our attacker model. In Chapter 4 we present related work on PSI, enumeration attacks and hash reversal. We then present our solutions for brute-force approach in Chapter 5 and lookup database in Chapter 6. Last, we present our evaluation in Chapter 7 and conclude this thesis in Chapter 8.

2. Background

In this chapter we clarify the theoretical basis of phone numbers, hashes, two databases and rainbow tables. We first discuss the international standard of phone numbers and detail the inner workings of assigning phone numbers to users in Section 2.1. We discuss the basis of hashes and detail specifically the SHA1 algorithm in Section 2.2. Next, in Section 2.3, we describe the inner workings of the database Redis. In Section 2.4 we detail another database with LMDB. Last, in Sections 2.5 and 2.6 we introduce the concept of rainbow tables with an implementation called RainbowCrack.

2.1. Phone Numbers

In this section we limit the amount of possible phone numbers by testing for their validity. This part is central to our work, because the phone numbers are used as an input to SHA1. So, by limiting the amount of numbers as input, we can then optimize the search for the inverse of hashes.

Telephone numbers were first introduced in 1879 to ease the work of telephone operators. Since then the format and definition changed frequently and also differed between countries. Until the ITU-T introduced the international public telecommunication numbering plan E.164 [20] in 1997.

E.164 defines (as seen in Figure 2.1), that phone numbers are internationally limited to 15 digits. This results in an input space of 10^{15} numbers and therefore hashes to be searched/saved. The first obvious limitation to this vast space is, that the first three digits are reserved for country codes. Another limitation is done by specific countries. For example, the North American Numbering Plan (NANP) doesn't set aside specific prefixes for mobile numbers. In contrast, most other countries offer prefixes specific to carriers. For example, Germany uses prefixes for specific mobile carriers [21]. This fragmentation allows us to limit our search space to a fraction of 10^{15}.

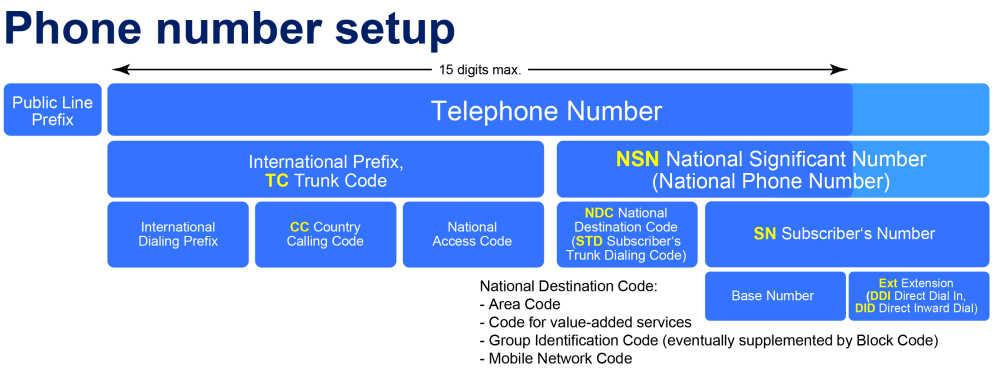


Figure 2.1.: Phone number setup [22]

In order to do so, numerous online services exist to test the validity of phone numbers. As seen in our work [2], we were able to leverage Google's libphonenumber [23] to filter out invalid numbers. This service is most prominently use by Signal [24]. By querying all prefixes with Google's service, the search space was reduced to around 353 billion numbers/hashes. Further, using WhatsApp's online validation system, we reduced our search space to roughly 118 billion possibilities.

So to summarize, in our work [2] we were able to first generate a — country code based — prefix database. This was further branched into country's specific prefixes. For example only valid mobile providers were considered for Germany, such as T-Mobile (+49151), Vodafone (+49152), etc. Then we tested the accepted length of those prefixes with libphonenumber for a maximum and minimum. For example, the number space of +49160 can have seven to eight additional digits after the prefix. To further reduce the number of possibilities we tested them against WhatsApp's registration/login API. As a result we reduced our search space by a factor of 10^4, allowing us to reverse hashes back into phone numbers within a realistic period.

2.2. Hashes

Secure Hash Algorithms (SHA) are cryptographic hash functions provided by National Institute of Standards and Technology (NIST). These algorithms are in our use-case used by services like WhatsApp and Signal to obfuscate phone numbers of users from their own servers. To be able to evaluate the privacy implications of using SHA as obfuscation, we need to understand their inner workings. We therefor provide a detailed description of SHA1 and a comparison to other algorithms of the SHA-family in this section.

Hash functions first and foremost map a variable sized input to a fixed sized output. This implies the possibility of collisions, as the input space is normally bigger than the fixed output space. The central point of defining a hash function is to minimize the possibility of collisions to a negligible amount. If the probability of collisions is negligible for a given function, this function is then called collision resistant.

As for our scenario, we not only need a collision resistant function, but also a one-way function. For that, a function needs to be practically nearly impossible to inverse. These are then called cryptographic hash functions, which we refer to in this work simply as hash functions or hashes. Cryptographic hash functions normally have more properties than mentioned (for example pre-image resistance), but are further not discussed as they are not relevant for our scenario.

As mentioned before, we are focusing on the SHA-family, more specifically on SHA1. This algorithm was first defined by NIST in FIPS 180-1 [25] and then extended with an open C

implementation as RFC 3174 [26]. SHA1 itself is a Merkle-Damgård construction [27], so it is based on a one-way compression function (maps two same-sized inputs to same-sized output as in Figure 2.2) and a preceding padding procedure. The padding is needed to extend the input to a multiple of the input for the compression function.

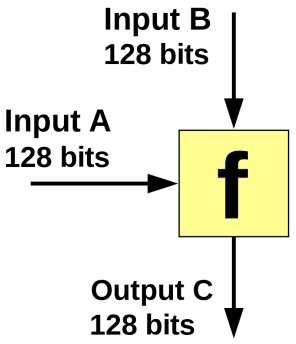


Figure 2.2.: One-way compression function [22]

In the case of SHA1, the message is padded to a multiple of 512 bits. First the bit '1' is appended to the message, then the message is appended by '0's leaving enough room for a 64 bit long integer, which represents the size of the original message. The result is then broken into 512 bit chunks, feeding SHA1's compression function. This function is based on a fixed initialization vector (IV) and reiterated 80 times, using four distinct and altering functions to generate and updated IV. This updated IV is then used for the next chunk. Once every chunk of the message is processed, the last IV becomes the message digest.

While using a very similar algorithm to SHA1, SHA2 mainly introduces specific constants for every round in the compression function. The chunk size is either 512 or 1024 bits and has either 64 rounds or 80 rounds - depending on the SHA2 variation at hand. Most notable the output size is increased and offers therefore a bigger range.

In contrast to SHA0, SHA1 and SHA2, which are based on a Merkle-Damgård construction, the newly introduced family of SHA3 (also called KECCAK) algorithms utilizes a so-called sponge construction [28]. Sponge constructions avoid several security flaws of Merkle-Damgård construction – like multi-collisions [29], length extension [30] or herding attacks [31].

2.3. Redis

Redis [16] is a key-value database, which resides in-memory. The basic idea behind this database is to save key-value pairs in the computer's Random Access Memory (RAM). This setup allows for very fast look-up times, meaning searching for a specific value by its key is done in constant time. In Redis' case the time is $O(1)$ in Bachmann–Landau notation [32]. But in return Redis' data is volatile, meaning all data is lost after a reboot.

Redis achieves those fast look-ups by implementing so-called memory hash tables [33]. Those hash tables utilize the RAM, as instead the sequential nature of disks by MySQL [34]. In principal hash tables are based on hash functions, which map a key to a so-called bucket in which the value is put. To exemplify this process we can look at Figure 2.3, where the names of contacts are keys and the values in the buckets are phone numbers.

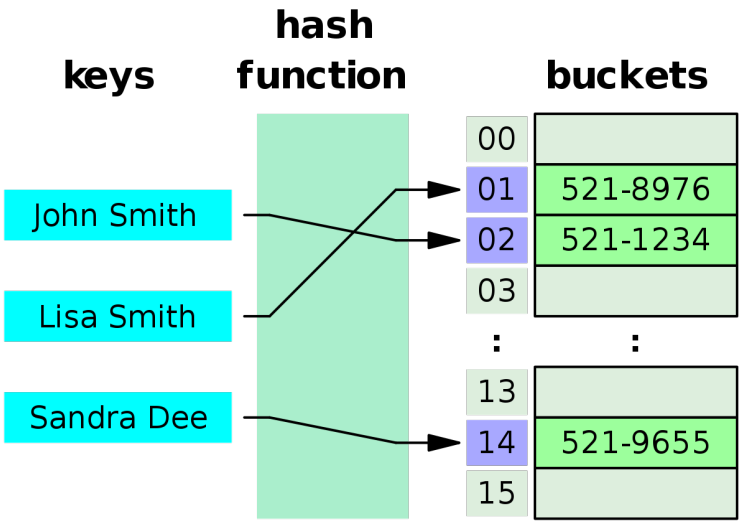


Figure 2.3.: Hash table example as phone book [35]

Hash tables don't initially map their inputs/value to every possible key of their hash function. For example, Redis uses the djb2 algorithm as a hash function. This function maps a string to a 32-bit digest. In theory, one can generate 2^{32} buckets with that function. But in practice one needs only a portion of buckets. In that case, we use the modulo operator to leverage remainders.

$$\begin{aligned} \text{digest} &= \text{djb2(key)} \\ \text{index} &= \text{digest} \mod \text{table\_size} \end{aligned} \tag{2.1}$$

If we use Equation 2.1 as a reference, we can see, that a total of table_size buckets are in the hash table. While the number of buckets can be sized as desired, by choosing the size as a power of two, we can leverage the faster bitmasks instead.

As we have mentioned before, Redis uses the djb2 hash function. This function is not a cryptographic hash function. In the particular use-case of hash tables, we need a hash function, that has a uniform distribution instead of high randomness. In other words, we need a hash function, that fills every bucket first before a collision occurs.

As the matter of collision arises, this is an expected circumstance in hash tables. As the birthday problem [36] indicates, there are going to be two keys assigned to one bucket. Several techniques exist to mitigate this problem. We focus on the one, Redis employs – as it is the most relevant for us to consider. Redis uses separate chaining with linked lists. So instead of just saving one value to every bucket, Redis saves colliding keys and their values as linked lists in their respective buckets. This results in lower lookup times, if the number of values per bucket is too high. Again, this worst-case scenario heavily depends on the chosen hash-function.

In Equation 2.1 we defined the size of the hash table as a constant. This would mean, that we need to know the size of the database beforehand. Normally this is not the case. Hash tables can therefore be resized. Redis uses so-called incremental resizing. The idea behind this technique is to first allocate a new (bigger) table. Then inserting every new entry into the new table, while also moving a small amount of keys from the old table to the new one piggybacking every new insertion. After all old entries are moved to the new table, the old one can then be deallocated.

This is obviously a big computational effort, especially in our case of inserting a large amount of entries. As we are looking for look-up times of inverting hashes, we don't emphasize this part of our setup.

The complexity of look-ups with Redis is $O(1 + \frac{n}{k})$, where n is the number of values and k the number of buckets. This fraction is being kept as low as possible by Redis. With resizing their table size according to the number of entries, Redis achieves an amortized complexity of $O(1)$ with many entries.

As mentioned in Section 2.1, we need to store roughly 118 billion phone numbers and their hashes. But the hash function djb2 allows "only" 2^{32} keys – roughly 4.3 billion entries to Redis. Luckily Redis offers an internal cluster functionality. This allows us to scale Redis to much more entries. The clustering offers 16384 different slots to make use of more instances of Redis. The process is rather simple. Redis takes the CRC16 of the key modulo 16384 and assigns the key to the corresponding slot. This slot then references to a predefined Redis instance (on the same server or remote) and is saved there. In return, if we search for a key, we just take the CRC16 of the key and "find" the corresponding server in constant time.

To mitigate volatility of data, Redis saves the data continuously on the disk. This obviously costs resources, such as disk space and computation time. In addition to saving data constantly on the disk, this procedure bottlenecks insertion time of data into the database.

But in our setup we can assume, that a continuous save on disk is not needed. We argue, that the database should be measured in a fixed state. So once every entry is written into the database, an attacker doesn't need to extend the database constantly. In other words, once the database is filled, Redis won't write data back to disk, as no data is saved any more. Therefore we reduce our setup-time, but don't skew with our read measurements. So we disabled this feature to re-insert our data in several stages of the evaluation.

2.4. LMDB

LMDB is key-value database designed as a library and written in C. In contrast to Redis, the LMDB stores its entries on disk rather than in volatile RAM. As a result the basic design is very different to the concepts discussed in the previous section. Instead of using hash tables as a data structure, LMDB relies on so-called B+-trees. In addition to this structure, LMDB maps its storage file into the virtual address space, which yields faster lookups as I/O operations are reduced.

We don't want to offer a detailed introspection of LMDB in this section, but rather a basic overview of the fundamental design choices of this database. We discuss B+-trees, memory-mapped files, append-only design, Multiversion Concurrency Control (MVCC) and copy-on-write.

B+-tree. A B+-tree is tree-like sorted and searchable structure, which holds values, that can be searched using search-keys – similar to Redis. In this context keys are not yet clearly defined. The keys can be variable length and type. One can think of a phone book as a key-value store, with names as keys and numbers as values. Names in a phone book can be variable length and can consume a different amount of space. In our case we define keys as SHA1-digests.

The tree consists of three types of nodes: one *root* node, *internal* nodes and so-called *leaf* nodes. The *root* node is normally considered an internal node, but can initially be a leaf node – if no other nodes are present. *Internal* nodes consist of a predefined amount of placeholders for keys and references to leaf nodes, but no values (nor pointers to them)

are saved here. A *leaf* node contains also keys, but holds pointers to the values on the disk and a pointer to their neighboring leaf node. All leafs are on the same level/depth of the tree.

The view of a B+-tree is defined by its order b. This number defines the number of children each node can have. For each of the three node types (root, internal and leaf) presenting the tree, there is a specific maximum and minimum number of children present. Children in this context refer to either records stored in the nodes (leaf or root node) or the amount of keys/pointers to nodes below the current node (internal node). Please be aware, that pointers are also counted as children. So every node, which can have b children, only contains $b-1$ keys (more in the example below). This can also be true for leaf nodes, as the pointer to the neighboring leaf node is also counted – depending on the implementation and visualization.

By design a root node as a leaf can have at the minimum 1 child and maximum $b-1$ children. If a root node is considered an internal node, then the node can hold the minimum of 2 children and at maximum b children. For every other node, the amount of children is uniform. Every node must have at least $\lceil \frac{b}{2} \rceil$ and at the most b children.

A B+-tree is balanced, meaning the distance from root to leaf is the same for every node. This is achieved by "rebalancing" the tree on an input depending on the current keys inside a node. We explore this in more detail with our example below.

We now look at an example of a growing B+-tree. More precise, we look at the process of inserting values into a B+-tree. Other processes are not the focus. For example, deleting an item is not important to our use-case, as we want to have potentially every mobile phone number in our database.

Let us consider the following ordered input of our example: $\{20, 9, 8, 3, 4, 5, 6, 33, 30, 44\}$. The B+-tree has the order of 4. We see in Figure 2.4 the initial state of the tree containing the first inputs $\{20, 9, 8\}$. Please notice the ordered structure of the node.

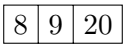


Figure 2.4.: Initial B+-tree

If we then include the next input $\{3\}$ into our node, the node is overfull and needs to split into two nodes. For even numbers, the split is done exactly in the middle - like in our example. If the order is odd, then the split depends on the implementation – if the key in the middle is assigned to the left (existing) or right (new) node. We can see the result of our split in Figure 2.5. We can also see, that the parenting node holds the first key of the right node. This is done, so the new key can be compared to the current keys of the node. If the new key is smaller, then the internal cursor moves to the left node, otherwise to the right.

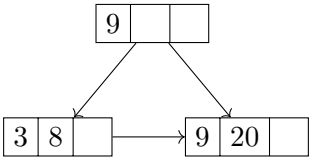


Figure 2.5.: In between step B+-tree

We now include the rest of the inputs $\{4, 5, 6, 33, 30, 44\}$ in Figure 2.6. We can first see, that the root node has now 4 children. If we search a key now in this node, we need to look, in between which of the current key-values our key fits and then go to the corresponding child node.

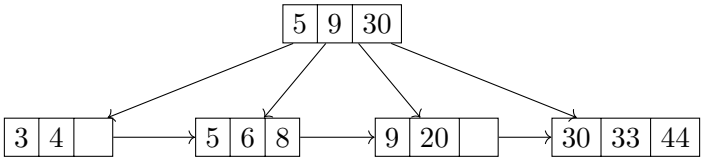


Figure 2.6.: Final B+-tree

The previous example figures omit the pointer to actual data in the leaf nodes.

LMDB B+-trees are obviously flattened, as the structure is held by atomic arrays. An interesting point to mention is, that LMDB doesn't save the pointers to neighboring leafs, but instead traverses back to the parent and just takes the pointer of the next key in the node.

In LMDB the user also doesn't define the order of the tree. The tree adopts dynamically to the inputs. One can consider, that in a generic database input values and keys are not uniform. Due to this generic nature of databases, the order (i.e. how many keys/values per node) is depended on the atomic block size – in our case page sizes of the operating system. In fact, LMDB's documentation urges users of the database to align entries according to page sizes and have the database size a multiple of the page size.

Memory-mapped files. Normally when opening a file within a program, we use so-called file descriptors to access files on disk. In C, this is done with the function `fopen`. This a wrapper provided by some libraries to actually call the system call `open`, handing further computation over to the Operating System (OS) until the file descriptor is passed to the running program.

In contrast to this approach, OSs also offer so-called memory-mapped files. Instead of handling file descriptors, the OS maps the entire file into the virtual memory space.

This used to be a rather limiting factor, as for 32 bit architectures, the maximum size of those files is limited to 4 GiB. In recent years, 64-bit architectures have become the dominating architectures. Nonetheless, most of those Instruction Set Architecture (ISA)s implement only a 48 bit wide address space, this still allows memory-mapped files to hold enough space for most database use-cases – around 128 TB [37].

The most beneficial factor of using memory-mapped files is the lack of system calls usage. While system calls are a crucial construct in OS to userland relation, system calls are costly in terms of computation time. By limiting the use of such functions like `open`, memory-mapped files achieve faster read and writes on large files.

Some drawbacks are, that memory is handled in page-sized chunks. So if the file is not aligned to OS's page size, some space is wasted. Also page faults can occur, in which case the OS needs to write the file to memory, resulting in further I/O penalty. Memory-mapped files also require the presence of a memory management unit (MMU), which is not given on every piece of hardware.

Some of those drawbacks are handled internally by LMDB. For instance, LMDB tries to align the database and data entries to the page size of the OS. Also LMDB inserts new data without overwriting or moving existing data on the disk. Hashes **Append-only design**.

As the name indicates, this database design dictates to only append new data – in fact generating an immutable data structure. This design allows to keep record of a stream of input events, without the need to implement explicitly more complex principals (i.e. data integrity).

While this is used in some applications (blockchain, sensor data, some databases), LMDB doesn't follow this design entirely. By using the append-only design, the database would necessarily balloon over time.

In order to keep the size of the database in check, LMDB actually deletes entries from its database. At first, this would lead to a fragmentation of data. One solution adapted by other implementations is to move data into the emptied space and recalculate the entire tree, which is very costly. Instead, LMDB implements a second B+-tree to keep track of those deleted data pages. If – and when – new records are added, LMDB uses those free pages to save the new records.

We see in Figure 2.7, how the old root with its updated entries F and G still exists. In an append-only design, this will eventually fill up the space. We can see LMDB's solution in Figure 2.8, where one tree is keeping track of unused pages, while the other reflects the current tree.

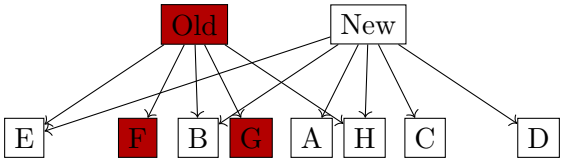


Figure 2.7.: Appended version of B+-tree

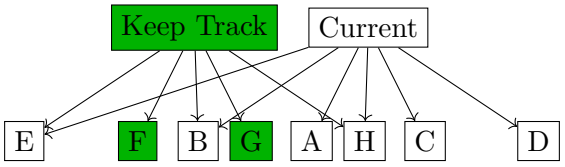


Figure 2.8.: Update version of B+-tree

Copy-on-write (COW). COW is a technique to manage resources, in our specific case the memory-mapped file. While other resource management techniques rely on some complex structure like multiple mutexes, COW only copies data segments once a write occurred.

For example, LMDB allows concurrent reads on the database. The amount of readers is not limited by the design of LMDB (i.e. locks), but other factor (bottlenecks like RAM, number of processes, etc.). While readers can create a transaction to read data, only one writer is allowed.

Even during a write transaction, readers' transactions are completed on a valid database before the write occurs due to the writer's COW. This guarantees data integrity and allows simultaneous transactions by readers.

Once the write is done, the readers shared memory is updated to the new tree – if no transaction is currently open. This is done by the underlying OS, as memory-mapped files are handled by it. The consistency is guaranteed by the append-only design – no actual data is overwritten.

LMDB's COW design has one major drawback. Once multiple readers have opened several pending transactions without closing them, the database is in an only-append frenzy. The older pending transactions need to have an old view of the tree, so LMDB can't overwrite newly deleted data. As a result, the space is eaten up, until transactions are closed and unused pages can be reused by LMDB's double tree design.

Multiversion concurrency control (MVCC). As mentioned above, LMDB allows users of the database to scale the number of readers linear, while the write-transactions are serialized. This means, that multiple programs can read the database concurrently, but only one write is done at a specific point in time.

In general, MVCC allows the database to hold multiple versions of a database. LMDB keeps therefore track, what data needs to be kept for every open transaction. This is done in LMDB by the concepts discussed before.

In LMDB the structure is kept in memory using a memory-mapped file. This type of shared memory is handled by the OS. While multiple transactions are open, the append-only design and COW allow to write to the database without changing the updated memory pages. Therefore keeping multiple versions of a database "alive".

2.5. Rainbow Tables

The idea of rainbow tables was published in 2003 as an optimization of previous work [19]. The basic concept of rainbow tables relies on the fact, that hash functions normally don't change. As an example, one can look at the handling of passwords in legacy operating systems like Windows 95, where passwords on every machine are hashed in the same manner. In other words, if two different users have the same password, the calculated hashes are identical on both machines. With that knowledge an attacker can precompute hashes with various password candidates as inputs and share those tables with others to facilitate inversion of stolen hashes. The resulting database can be very large though, and rainbow tables is a technique to reduce the storage requirements at the cost of additional computations.

Rainbow tables try to minimize the needed space by chaining hashes together. This is done by using the concept of hash chains [38] with further optimizations. The hash chain is a sequence of hash values produced by alternately applying two functions. One is the hash-function, that is being attacked. The other one is a so-called reduction function, that is designed to map a hash back to the character space of potential passwords.

Now let H be a hash function, which produces a 32-bit digest. Let R be a reduction function, which maps hex-values to 8 digits. As input for the hash function, we use the same charset as the R's output. We can then build a chain that looks like the Figure 2.2.

$$00000000 \underset{H}{\longrightarrow} \texttt{000FF1CE} \underset{R}{\longrightarrow} 12345678 \underset{H}{\longrightarrow} \texttt{CAFED00D} \underset{R}{\longrightarrow} 13141516 \tag{2.2}$$

We only save the starting value 00000000 and the end 13141516. So, if we are trying to reverse the hash `CAFED00D`, we first reduce it and look it up. If we find a match, we can just recompute the chain and find out, that 12345678 was the input to generate `CAFED00D` with H. If we don't find a match, we start traversing the chain back, by applying again first R, then H, then R again. If there is still no match, we add another cycle of reducing and hashing, depending on the length of each chain. This is done until a match is found or the table of hash chains is exhausted.

There are some pitfalls to look out for. Naturally we can't alter the chosen hash function, but the reduction function(s) need to be carefully crafted. If we look at our example in Figure 2.2, we can extend this table further with the Figure 2.3.

$$12121212 \underset{H}{\longrightarrow} \texttt{DABBAD00} \underset{R}{\longrightarrow} 12344321 \underset{H}{\longrightarrow} \texttt{BADDCAFE} \underset{R}{\longrightarrow} 13141516 \tag{2.3}$$

We can see in Figure 2.3, that the last value is the same as in Figure 2.2. In other words, R produces the same output with the inputs `BADDCAFE` and `CAFED00D`. So if we search for `CAFED00D` like before, we would hit a match in the chain in Figure 2.3. But we can't reverse the hash, as `CAFED00D` is not represented in this chain. This is called a false alarm and is just skipped over.

Another problem arises with false alarms. While the attacker might not find the inverse of the hash in a row that matches, the inverse can still exist in other rows. But if we look at Figure 2.4, we can see that the chains may collide and merge.

$$\left.\begin{array}{l} \ldots 00000000 \underset{H}{\longrightarrow} \texttt{000FF1CE} \underset{R}{\longrightarrow} 12345678 \underset{H}{\longrightarrow} \texttt{CAFED00D} \\ \ldots 12121212 \underset{H}{\longrightarrow} \texttt{DABBAD00} \underset{R}{\longrightarrow} 12344321 \underset{H}{\longrightarrow} \texttt{BADDCAFE} \end{array}\right\} \underset{R}{\longrightarrow} 13141516 \underset{H}{\longrightarrow} \ldots \tag{2.4}$$

Those collisions are very problematic, as they cost an attacker computational time, space and lookup-time. In our example the collision seems to happen at the same position of the chain, but that is ordinarily not the case. Most likely the beginnings and endings (the only things saved) of overlapping chains will be different, because the collision of the two chains is most likely on different positions in their respective chain – as the chains have different starts. Thus the collision can't be easily identified by looking through the table.

Mitigating those collisions is, what Rainbow Tables [19] improved over the hash chains. The difference to simple hash chains is the introduction of different reductions functions per "column". In fact, by defining the length of each chain as $k \in \mathbb{N}$, we also need to have k reduction functions, i.e. $R_1 \rightarrow R_k$. This change doesn't stop reduction functions to map to the same output. We can consider the following two cases. First, the collision happens in the same column, so $R_i = R_i$ where $i := 1 \ldots k$ in both colliding chains. In that case the last reduction with R_k constructs the same output, which is saved in the table. We can just scan the rainbow table for double entries and generate new chains. Second, if the collision is at the position $R_i \neq R_i$ where $i := 1 \ldots k$, the following reduction functions are also not identical and produce eventually different outputs. As stated before, collisions still happen, but are greatly reduced and lead to overlaps not entire merges.

While the basic construction of Rainbow Tables is similar to hash chains, we need to be careful doing the lookup. Instead of just altering R and H, we need to follow the order of different Rs. Let's look at the similar Figure 2.5 as before, but with different Rs.

$$00000000 \underset{H}{\longrightarrow} \texttt{000FF1CE} \underset{R_1}{\longrightarrow} 12345678 \underset{H}{\longrightarrow} \texttt{CAFED00D} \underset{R_2}{\longrightarrow} 13141516 \tag{2.5}$$

Let's search for `000FF1CE`. We start by applying R_2 to our hash. This for example results in 12341234, which is not in the "table". After that we use R_1, then H and last R_2. As a result, we have a match with the last value, which is saved in the table. From then we start recomputing the chain, until we have the inverse to `000FF1CE`.

To put this into more formal terms. We start searching by applying the last reduction function R_k. If we don't have a hit, we start with the previous reduction R_{k-1}, then applying H and R_k. We retrace the chain, until we have a hit, or the length k is exhausted.

As mentioned before, this kind of attack is based on the notion, that the hash function doesn't change. In other use cases, like passwords, this attack vector is made infeasible by applying a so-called salt with the hashing algorithm. With salt inclusion in the hashing process, even if two passwords are the same, the calculated hashes are different due to the salt. So, an attacker would need to calculate a Rainbow Table for every possible salt. In reality, salting hashes made using Rainbow Tables impractical for real-world attacks on hashes.

However, salted hashes are ill-suitable for the use case of Contact Discovery. If two users with the same phone number of a mutual friend try to find this friend, the service can't match them by using different digests of that phone number. Hence, it becomes possible to apply Rainbow Tables as an attacking technique in this case.

2.6. Rainbow Crack

In this section, we describe the usage of a modified RainbowCrack-NG tool [39]. The original tool is an implementation of Phillipe Oeschlein's Rainbow Tables [19] and is modified from version *1.2.* for the hash reversal of phone numbers.

The implementation of the modified Rainbow Crack was done by Daniel Schindler in his practicum [18]. He programmed a novel reduction function specifically targeting mobile phone number reversal as part of [2]. The reduction function was designed in such a way, that calculated hash digests are reduced back to a valid phone number. With this optimization they were able to speed up the performance and lower the size of the table by a magnitude – compared to a straight forward use of Rainbow Crack.

Generation of Rainbow Table. Rainbow Crack is designed in such a way, that this process is separated into two distinct tools. One is called *rtgen*, which generates the rainbow table. The tool is called with six arguments:

1. Hash Algorithm (i.e. SHA1)
2. Index of Rainbow Table
3. Chain length
4. Chain count
5. Number prefix
6. Output file

```
./rtgen sha1 1 1000 1000000 DE_mobile out
        1.   2.  3.    4.        5.     6.
```

Listing 2.1: Example rtgen call

The hashing algorithm depends on the targeted application. In the case of Contact Discovery, we focus on the SHA-family, more specifically SHA1. The index of the rainbow talbe is used, in order to generate multiple tables targeting the same input space. This means, we can and need to generate multiple tables targeting the same input. The chain length defines the amount of alterations between hash function and reduction function. The chain count allows us to control the number of entries per table. With the number prefix, we can target specific phone numbers of countries, or generate tables against world-wide mobile phone numbers. The output file just defines the name of our output directory.

While we define the initial name of our output, Rainbow Crack adds all the information into the file name. An example is shown in the Listing 2.2. Here, all the arguments from Listing 2.1 are directly displayed.

```
./out/sha1_1_1000x1000000_DE_mobile.rt
```

Listing 2.2: Example file name

The other tool introduced by Rainbow Crack is called *rtsort*. To sort the Rainbow Table with this tool, we only need to hand over the file-name of our generated rainbow tables. This is simply done with the command in Listing 2.3.

```
./rtsort ./out/sha1_1_1000x1000000_DE_mobile.rt
```

Listing 2.3: Example rtsort call

Rainbow Table. In the case of Rainbow Crack, this component of the cracking process is just a file. The interesting part here is, that only the index of the clear-text is saved, but not the clear-text itself. The index has the size of a 64 bit integer, so Rainbow Crack saves only 128 bits per chain.

However, we not only use one file, but generate multiple tables with different indexes over the same input-space. This is done to accommodate for collisions. So, by generating multiple tables, the probability increases, that we cover the entire included phone number space.

Cracking. To crack hash digests we simply use the tool *rcrack*. This allows us to specify a table and the *Search Request*. The request can be interpreted in three different ways by Rainbow Crack:

- -h
- -l hash_list_file
- -f pwdump_file

The first flag *h* allows the attacker to search for one raw hash. This can be useful for us, but generally we need a way to search in bulk. This is facilitated by the flag *l*, where an attacker can specify a hash digest list. In this list, all digests are just separated by a new line. The last flag *f* is specifically to a tool called Lanmanager. Here, an attacker can directly include a password-dump file from that software.

All in all, we mostly use the *l* flag to search bulk requests. As mentioned before, in the context of Contact Discovery an attacker or malicious provider gains access to multiple hash digests of a victim/user. It's generally only processed as a bulk.

We see in Listing 2.4 an example call of *rcrack*. Here we search our previously generated rainbow table for the digests in the referred file. All other definitions are read directly from the file name of the table. In our case *rcrack* parses the name and gets the hash algorithm, the index, chain length, chain count and the used prefixes. Especially the last part is important, as the reduction function is based on the predefined prefixes used to generate the table.

```
./rcrack ./out/sha1_1_1000x1000000_DE_mobile.rt -l ./digests
```

Listing 2.4: Example rcrack call

For the random generation of a Search Request, we use the provided tool called *hashgen*. It uses the number prefixes to calculate specific digests based on phone number country codes. We can see in Listing 2.5 an example call. The first argument is the number of digests. The second is the number space based on the number prefix. The last part only saves the digests into a file called *hashes_DE_mobile*.

```
./hashgen 10000 DE_mobile > hashes_DE_mobile
```

Listing 2.5: Example hashgen call

3. Problem Statement

Contact Discovery is an essential part of mobile messengers. It allows users to discover, who among user's contacts are already registered with those services, or to invite those who aren't. In order for a service to distinguish registered and non-registered users, the service needs to compare the contact list of the user with an entire database of all registered users. This implies, that either user needs to send all the contacts to the messenger service, or vise versa. Usually, the former approach is used since it is more efficient (users have fewer contacts) and since in this way the service provider does not reveal information about all the registered users. Once all contacts are uploaded the service compares the two lists and sends back the user an intersection between registered users and contact's users.

This basic idea for Contact Discovery is used by virtually every major mobile messenger app, including WhatsApp, Signal and Telegram. At first glance, the solution is pretty straight forward, but implies a major privacy concern: The service providers get the knowledge about all the users' contacts – including those who aren't registered with the service. This fact enables the providers to generate a social graphs of users, including people, who are not registered.

While those graphs are intrinsic to all types of social networks, there is a general push by mobile messengers to provide a more privacy oriented design. In the following, we discuss privacy-preserving measures employed by eight mobile messengers: Telegram, WhatsApp, Snapchat, Threema, WeChat, Viber, Wire and Signal.

3.1. Telegram

Telegram was released in 2014 and aims to provide a cloud-based messenger service. As of April 2020 Telegram has over 400 million users worldwide [40]. Telegram uses phone numbers as the primary source of registration of an account, while allowing the use of multiple devices with one account.

Phone numbers are seen as personally identifiable information in most countries, so Telegram can't be considered anonymous due to their need for registration by phone number. Most users try to gain already anonymous phone numbers (i.e. burners) to facilitate anonymity within Telegram.

Telegram uses its own cryptographic protocol called MTProto. This decision is widely criticized as it is an unvetted protocol, which could potentially include undiscovered flaws.

MTProto uses AES as symmetric encryption and a Diffie–Hellman key exchange to enable encryption between Telegram's servers and clients.

Open Source. Telegram offers a public API to access their service, so most of the official client-side apps are open-source and can be reviewed for malicious behavior. While Telegram promised to eventually release the source-code of their server-side architecture, up to date there is no official source of information to identify, what is running and stored on their servers.

Anonymization. We have to concentrate on the third-party clients and Telegram's official API to evaluate their privacy methods in Contact Discovery. As defined in Telegram's official documentation of their API [41], a list of contacts is sent to the server without any obfuscation of phone numbers, names or other data.

We can also see a direct transfer of contact data to Telegram's server in third-party clients [42]. This is a privacy concern of every user and even worse, those contact data are sent to Telegram's server without their permission.

Identification. As mentioned before, Telegram uses solely phone numbers for the identification of accounts. So most Contact Discovery is also done via phone numbers. Nonetheless, Telegram allows user to define a username, which allows other users to search by usernames.

3.2. WhatsApp

WhatsApp is a popular messenger app and was initially released in 2009. By end of 2013, WhatsApp reached over 400 million monthly users [43]. In February of 2014 Facebook acquired WhatsApp and continued its services [44]. As of February 2020, WhatsApp now has over two billion users world-wide [45] and over five billion installs from Google Play Store [46] – making the service the most popular messenger world-wide.

Open Source. Unlike Telegram, WhatsApp offers no source code neither for its client nor servers. This makes it generally unclear, what measures are taken by WhatsApp to ensure the privacy and security of its users.

There are some efforts to reverse engineer WhatApp to build open clients [47] or to manipulate chats [48]. While those efforts exclusively focus on the client-side of things, they provide insight to the actual data sent to WhatsApp – thereby validating some of their claims.

Anonymization. In contrast to Telegram, WhatsApp mitigates the privacy problem of Contact Discovery by hashing all phone numbers before sending them to their servers [49]. While storing hashed phone numbers seems at first sufficient, it doesn't provide much more privacy for the user.

In an anecdotal story [50], WhatsApp allegedly shared social graph information with its parent company Facebook. This resulted in Facebook recommending patients of a psychiatrist to other patients of hers. This story can even be true, if phone numbers are stored in a hashed form. One can still infer social contacts by comparing mutual hashes. Those mutual contacts allow then to link two persons, without them actively using this service.

So the last apparent privacy gain of hashing phone numbers is, that WhatsApp doesn't store the actual numbers, but only has a non-identifiable hash. As we will further discuss in this thesis, even this gain is negligible, since we show it is possible to efficiently reverse hashes based on mobile phone numbers.

Identification. WhatsApp bases its identification mechanism solely on mobile phone numbers. They even omit devices, that are not attached to a mobile phone number, such as

iPads or iPods [51]. Nonetheless, they offer a web presence, which can be used by scanning a QR-Code from the WhatsApp app [1] – indirectly still relying on phone numbers.

3.3. Snapchat

Identical to WhatsApp, Snapchat's parent company is Facebook, but uses usernames to identify different users. Snapchat is a social media platform for sharing short-lived photos, reaching over 229 million users [52] in 2020. The service also allows to exchange messages with other users – making it a mobile messenger.

Open Source. Snapchat hasn't publicized their source-code like WhatsApp. Similar to other closed-source services in this chapter, Snapchat offers an API [53]. While APIs generally allow users to extend the functionality of the app, they don't offer much insight into Snapchat.

Anonymization. As part of their Contact Discovery Snapchat offers users the possibility to upload their contact lists to find their friends on the platform. In their privacy policy [54] they clearly state to use contact lists of other users to infer more information about those users. Unfortunately Snapchat doesn't spell out, how the contact lists are transmitted or in what capacity these lists are stored and shared with other Facebook companies.

Identification. Snapchat identifies their users by usernames. The registration steps only require to sate a username and password. To regain access to an account, Snapchat offers to also register an email address or a mobile phone number [55].

3.4. Threema

Threema [10], another messenger app, shows, that one can take a vastly different approach. This app removes the necessity to share personal information and relies on eight digit IDs to have an unique identifier for every user. Threema is proprietary, so similar to WhatsApp we can't validate their source code.

While most other apps on this list are freeware, Threema costs 3.99 € and has over one million downloads in the Google Play Store [56]. According to their press release, Threema has over eight million users as of January 2020 [57]. One major jump in user numbers was seen after Facebook took over WhatsApp – indicating a valid user-base in demand for private mobile messengers.

Open Source. Threema doesn't share its source code, neither for clients nor servers. The authors instead offer a whitepaper explaining their high-level architecture [58]. In this document the authors describe different deployed techniques to ensure the users safety and privacy. Clearly, this doesn't replace an open-source approach, but offers some more insight how the service is build.

In addition, some researchers reverse engineered Threema's API [59] and publicized an open-source implementation of the Threema protocol in Go [60]. The author gave also a talk [61], allowing the public to have some minimal overview of the actual implementation of Threema.

Anonymization. Threema falls here victim to usablity, despite their alternative approach. The application offers an opt-in for users to register their contact data (i.e. phone number or email address). Using this discovery model, Threema also relies on hashing those values and sending them for comparison to their servers [62]. Otherwise, the users would need to share their Threema IDs with other users using an out-of-band communication channel.

Identification. As mentioned before, Threema allows users to only use their eight digits ID. Otherwise the application allows users to discover contacts by their email address and/or

phone number. This form of discovery depends solely on the permission of the users. Nonetheless, Threema also processes hashed phone numbers of non-registered users, which inherently depends on trusting Threema due to their closed-source code.

3.5. WeChat

In 2011 WeChat was released by Tencent Holdings Ltd. By May 2018 the app gained over a billion active users [63] and is mostly popular in China. WeChat and its parent company Tencent operate under Chinese law [64], which implies by itself some privacy related concerns [65].

Open Source. WeChat is generally not open-source. However, the messenger offers a public API [66], which can be used to build third party tools [67]. Still – similarly to other closed-source messenger in this chapter – we can only speculate how the vast amount of data is further used to build wide-reaching social graphs.

Anonymization. Further WeChat collects the users contact list automatically [68] and apparently saves everything without anonymizing third-parties in those lists [69]. Therefore accessing personally identifiable information such as phone numbers of people, who are not using their services in cleartext.

Identification. WeChat handles identification similar to WhatsApp. The main focus is on mobile phone numbers, only allowing registration and identification with a valid number. They also provide a web-client, which can be accessed by scanning a QR-Code from their phone app [11] – thereby guaranteeing a valid verification per mobile phone number took place.

3.6. Viber

Another popular mobile messenger app Viber [12] was initially released in 2010. The messenger was acquired by the Japanese company Rakuten [70] [71]. Viber is mostly popular in Eastern Europe, reaching over 100 million users in Russia by 2018 and becoming the most downloaded messaging app in Ukraine by 2016 [72]. The service registered over a billion users world-wide in 2018 [73].

After some security concerns Viber copied Signal's encryption protocol, to boost their own privacy and security policies [74]. But the service applied no new privacy protocols in regard to their Contact Discovery via contact lists.

Open Source. Viber is not open-source. Similar to others in this chapter, the messenger offers an REST API for developers [75] – for example to create bots.

Anonymization. Viber automatically uploads the entire contact list of theirs users [76]. Even worse, it appears those data is not in any way anonymized and saved without any time limit [77]. This includes data of person, who are not registered nor using this services.

Identification. Identical to previously mentioned messengers, Viber uses phone numbers as the sole identifier. Though the service offers desktop clients, the users need to unlock them with the mobile app [78] – consequently still relying on personally identifiable information.

3.7. Wire

The privacy-focused messenger app Wire [13] was first released in end of 2014. A part of the initial team were previously involved with Skype [79] [80] – another messenger app. This expertise allowed the authors to focus on a security-by-design approach. In

this context they released two whitepapers, discussing the security [81] and privacy [82] of their application. In both papers, the authors discuss the high-level architecture of their application. For example, the authors detail their protocol for Contact Discovery or end-to-end encryption of clients.

While the current focus of Wire is to be a collaboration tool for businesses, they offer Wire as a personal messenger app. Their app has currently over a million downloads in the Google Play Store [83] – omitting Open Source builds and iOS versions of the client.

Open Source. Wire released their client source code under GPLv3 in 2016 [84]. A year later the server source code was also published on GitHub in 2017 [85]. They offer different target platforms for their client. Most prominently, they offer source code to iOS, Android, Desktop (based on Electron [86]) and Web clients – allowing users to inspect, alter and compile their code.

Anonymization. Similarly to other mobile messenger apps, Wire allows users to share their contact list with the service. After the user grants Wire access, the app hashes each phone number and then shares only the hashes with Wire's servers according to their privacy whitepaper [82]. Wire uses SHA-256 as the hashing algorithm to anonymize phone numbers. This approach is similar to Threema and relies on either exchanging app-id out-of-band or sending the entire contact list.

Identification. Wire allows the user to register an account, either by using an email address or a phone number. Both methods are only used to force a personal identification. Once the registration is complete, users can mostly be found via username or Contact Discovery based on the contact list.

3.8. Signal

In a sharp contrast to the previously mentioned messenger apps, Signal actively focuses on the issue of private Contact Discovery and pushes for new solutions in this field. Signal itself is a merger of two other systems called TextSecure and RedPhone and was initially released under this name in 2014.

As a step to ensure trust in the app, the source code for Signal's servers and clients was released, allowing the community to ensure the validity of their privacy and security claims. Another important part of Signal, is its open source end-to-end encryption scheme called Signal Protocol. This protocol is based on several well-known cryptographic primitives and algorithms and was formally proven sound in 2016 [87].

Open Source. Signal's predecessors TextSecure and RedPhone were released as open-source in 2011 and 2012 respectively. Signal itself was conceptualized as open-source from its beginnings. Another important release is the Signal Protocol, which is also publicly available and implemented by several other messengers, like WhatsApp, Skype and Facebook Messenger.

Anonymization. Signal initially aimed to provide privacy of thrid-party contacts by hashing all phone numbers with a truncated SHA1 (to save space) [88]. But this is not sufficient to guarantee privacy of those contacts, as we show in this thesis. Since then the service has published a private Contact Discovery based on Intel SGX [89] to guarantee private processing of contacts.

Identification. The main criticism of Signal is focused on the lack of diverse identification. Signal allows only to use the mobile phone number to register and identify to their service. As we described with the other messengers, a mobile phone number is considered personal identifiable information. Using such information is seen problematic by privacy advocates.

3.9. Attacker Model

In our case we focus on an attacker, who has access to the hashes of a service provider. We discuss potential attack vectors, i.e. how this access can be gained. The goals of the attacker can also be manifold. We describe several goals, but not all in this section.

Attacked data. The attacked data are hashed phone numbers. We focus on not just the user database, but also all data given to the service within the Contact Discovery process. This means, the attacker has the data of all users plus their phone books.

Attack vectors. An attacker can gain access through different methods. For one the actual service provider can snoop this information, even though they provide open source access. For another, the attacker can be an insider, who has some kind of access to the data – i.e. an administrator or developer. And last, an attacker can gain through compromise. This attacker can be a state-actor, ISP or a criminal organization.

Attack goals. The fundamental goal of all potential attackers is to reverse hash digests of phone numbers. The gained phone numbers can then be used in different scenarios. One can be to build extensive social graphs of users and using their phone numbers to identify parts of their social graph. This can be done by looking up phone numbers through different online services, like Facebook or phone books. Another scenario to sell phone numbers to telemarketers, or to use them in part of an identify theft.

3.10. Discussion

All in all every messenger app listed in this chapter uses at least partly mobile phone numbers to pair its users. We acknowledge that this form of Contact Discovery is a fundamental premise of these services, but – except for Signal, WhatsApp, Wire and Threema – no other service is even trying to mitigate this problem or to potentially solve it.

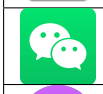
Another aspect is, that most services don't provide an open environment. Closed-source software can only offer limited privacy guarantees. A proprietary services can focus on the privacy aspects of their applications, but a user can't ensure, that those aspects are implemented in the closed-source software.

More crucially, only a few services allow to register and be identified without a mobile phone number. This is mostly done to provide a convenient way to find contacts on the service. While this is a user friendly feature, a privacy focused app like Signal forces its users to be identified by the mobile phone number.

By providing another identification method, a service can mitigate the privacy issues of Contact Discovery based on phone numbers. One could also define specific contacts to be discovered by the service. This would limit the exposure of user not registered with the service.

Millions, if not billions, of contacts are shared with those services, most crucially of people, who are not using those apps or agreeing with the company's processing of their personal data. We provide an overview of the different aspects summarized in Table 3.1.

Table 3.1.: App comparison

Logo	Name	Open source	Anonymizes	Other identification
	Telegram [90]	Server ✗ Clients ✓	✗	✗
	WhatsApp [91]	✗	✓	✗
	Snapchat [92]	✗	✗	✓
	Threema [93]	✗	✓	✓
	WeChat [94]	✗	✗	✗
	Viber [95]	✗	✗	✗
	Wire [96]	✓	✓	✓
	Signal [97]	✓	✓	✗

4. Related Work

In this thesis we have to consider multiple kinds of related work. For one, we look at the fundamental goal of Contact Discovery. In that, users try to find their friends by sharing their contact list with a provider like WhatsApp. This can be seen as an intersection of two sets – contacts of users and database of providers. Current research solves this problem by so-called PSI. We discuss related work of PSI in Section 4.1 and offer an overview.

For the other, we offer an overview of different techniques of hash reversal. In contrast to PSI, hash reversal centers around techniques against currently employed solutions mentioned in Chapter 3. In this context we discuss related work to brute-forcing hashes, rainbow tables and cryptoanalysis of SHA-Family.

Another approach to get the user-base of a service are so-called enumeration attacks. These attacks target publicly available information like APIs or username tests in order to enumerate lists by issuing multiple queries.

4.1. Private Set Intersection

PSI focuses on the problem of exchanging information. Imagine we have two parties, which want to compare their sets of items with each other without revealing all items of those sets. The only information given to the other party should be the result of the intersection to guarantee privacy. One can employ different techniques of cryptography to build a protocol to solve this privacy problem.

Generally PSI can be achieved in a naive way. If both parties just hash their items of the sets and compare the digests, both mutually only reveal the items of the intersection and preserve their privacy. Obviously the security of hashes heavily depends on the entropy of the input. As discussed in this thesis, the entropy of phone numbers is insufficient to guarantee privacy with this naive approach.

More complex PSI protocols are based on public keys (Diffie-Hellman, [98]), circuits ([99]), Oblivious Transfer (OT) ([100],[101]) or by introducing a trusted third party ([102]). Most of them make some kind of trade-off between computational and networking overhead or underpin trusted parts. Those limitations make them difficult to deploy in heavy-duty scenarios like WhatsApp or Signal.

There are multiple PSI protocols defined and suggested for different scenarios. Like the example above, both parties can hold a similarly sized set each, which we will categorize as

balanced PSI. If those sets are different sizes, we categorize those protocols into unbalanced PSI. After discussing those different approaches, we discuss a state-of-the-art technique solving our problem of Contact Discovery with some caveats.

Balanced PSI. In [103] the authors focus on OT-based PSI. While they achieve better performance than previous work, they limit the benchmark for sets lower than a million items. Also the authors only consider items of set with the size of four ASCII characters – a phone number can consist of maximal 15. Both limitations are exceeded by WhatsApp and Signal, with both having more than ten million users.

While more recent research like [104] offer better performance results, their test-set sizes are only around one million entries (2^{20}). Another limiting factor is the required traffic volume between the two parties. For bigger sets, the two parties need to communicate more often and generating.

Another factor to consider is, that balanced PSI aims to solve set intersection on two similarly sized sets. In our scenario we work with a user and a provider (i.e. Signal). The provider has clearly a more extensive set than the user. Balanced protocols therefore will offer poor performance with the problem of Contact Discovery.

Unbalanced PSI. In contrast to balanced PSI, unbalanced protocols aim to optimize for Contact Discovery on mobile applications. For example the authors of [7] define the size of item in the sets with 128 bits. This reflects an input of 16 ASCII-Characters, which fits the size of phone numbers. Also the authors generally assume a large set of the provider with up to 2^{30} items. However, the protocol needs a considerable amount of communication, which was optimized by later work.

In [105] and [106] the authors base their PSI protocol on homomorphic encryption. The principal idea is similar to our naive approach of hashing the items of the sets – but instead of hashes the authors encrypt the items. The protocol allows for a sufficient size of items, while also having a low communication overhead. Nonetheless, by encrypting the items this protocol has a substantial computational overhead.

State-of-the-art. The currently best PSI protocol for our scenario is the so-called ECC-NR-PSI [6] – also based on OT. This approach allows for a secure intersection of sets, without resulting in unreasonable computational performance on current smart phones. The authors implemented ECC-NR-PSI and ran tests with up to 2^{28} items in the set from the provider (i.e. WhatsApp). This is roughly the magnitude of users of Telegram. However, the protocol needs to transmit 1.07 GiB to every user requesting a new Contact Discovery. Obviously this limitation hinders an adaption of the protocol by current mobile messengers.

Summary. In conclusion, PSI protocols provide promising features, but are not yet applicable to large-scaled messengers like WhatsApp or Signal. While different protocols provide different trade-offs in respect to performance, none of the mentioned protocols can currently handle a real-world application like WhatsApp or Signal.

4.2. Enumeration Attacks

Enumeration attack is a technique to gain information by accessing specific interfaces. A well-known example of this attack are IP scanners. An attacker can try to access every possible IP address. Using only this access an attacker can infer which IPs are up, which ports are open and even a corresponding domain with name and address of the owner.

These kinds of attacks can also be transferred to mobile messenger apps and social networks. In contrast to the work in this thesis, enumeration attacks don't rely on another

breach or access to more data – like hashes. This technique uses only the Contact Discovery mechanism of providers.

Like every legitimate user, the attacker can request a Contact Discovery with their current contact list. Once the service sends back the intersection, the attacker can change her contact list and repeat the process. If there are no mitigations in place, the attacker can so retrieve all users of that service.

For example the authors in [107] were able to enumerate contacts of KakaoTalk. They used KakaoTalk's Contact Discovery to retrieve around 50.000 users by testing for 100.000 phone numbers. The service didn't use any noticeable mitigation like rate limits of requests, anomaly detection or a more restrictive model.

In another attack, researchers were able to enumerate around 25.000 facebook profiles from phone numbers [108]. The authors leveraged the fact, that anybody can search a profile by its number. While some countermeasures like anomaly detection were taken by facebook, the authors were able to bypass those by simply using multiple accounts.

There was also an enumeration attack on WhatsApp [109]. The authors were able to go through over three million phone numbers, without any noticeable limitation. Similarly to the previous attack on KakaoTalk, the attackers just changed their contact list and kicked off the Contact Discovery.

Some tools to enumerate contacts on WhatsApp are publicly available [110] [111]. While those tools may need to adapt to certain changes done by WhatsApp from time to time, they are still viable information gathering tools.

We also used an enumeration attack against WhatsApp [2]. Because WhatsApp implemented new mitigation techniques, we needed to first test for limitations and then secondly enumerate within those limitations. We were able to crawl 10% of all US phone numbers during 34 days and 25 accounts.

In general, enumeration attacks seem always in some way feasible. They repeatably misuse benign functions, which are central to the service. For some social networks the attack vector could be a simple search, which allows some information retrieval. Mostly those services block enumeration attempts by employing so-called CAPTCHAS or anomaly detection.

In the context of mobile messengers, the services use anomaly detection, but also hard limits on possible Contact Discovery requests. In some cases (i.e. Telegram) those limits are set so low, that an enumeration isn't leaking substantial amounts of numbers.

4.3. Hash Reversal

Hash reversal techniques are different approaches to find the preimage of a given hash digest. This is cryptographically only possible with a neglectable probability. To make those attacks more feasible, the attacker assumes, which input space was used to calculate the hash digest.

Those attacks were mainly created to gain plaintext passwords from a database, where the passwords were hashed. There are three main methods for hash reversal: Brute-force, rainbow tables and lookup databases.

Brute-force. This approach defines an input space for the hash function based on intuition. Then the attacker calculates every hash for this predefined space on-the-fly [15] [14]. After that, the attack-tool compares the generated hash with the provided database of given

hashes. If the calculated hash digest is found, the hash and its input will be saved. Otherwise the current hash is discarded and the next hash is calculated.

Rainbow Tables. Of course the previous method takes a lot of time and resources. In order to mitigate these drawbacks, Rainbow Tables make use of so-called hash-chains [112]. Here the attacker precalculates hash digests based on an input space. This digest is then reduced with a reduction function to the input space. Then the reduced input is hashed again. This process is repeated multiple times – generating a chain of hashes. In contrast to brute-force, the attacker saves the starting input and the last reduced hash.

This method allows the attacker to make a trade-off between the size of saved chains and the overhead of retracing the chain. In the context of passwords/databases rainbow tables are made obsolete with the use of salts. However, in the context of hash-based Contact Discovery rainbow tables are still applicable, because salts can't be utilized.

Databases. Using this approach, the attacker saves every calculated hash into a database with its input. Similar to before, the attacker first defines an input-space. Then every possible input is generated, hashed and saved. One can then simply query this database, once a new hash digest needs to be reversed.

In summary, all techniques mentioned above limit their input-space to make hash reversal feasible – which is already done in our case of mobile Contact Discovery. By limiting the input-space to only numbers, all attacks from above are useful and therefore discussed further in this thesis.

5. Brute-force Hashes

In this chapter we discuss our brute-force framework for the hash reversal of mobile phone numbers. We discuss first our general architecture in order to facilitate such an attack in Section 5.1. Then, we apply this architecture to specifically chosen tools in Sections 5.2 and 5.3.

The focus of this thesis is on reverting SHA1 hash digests in the context Contact Discovery. This means we want to compute the input to a given hash digest. For this task, an attacker can deploy a technique called brute-force. With this technique, the attacker calculates the hash digests on-the-fly.

In contrast to other techniques discussed in this thesis, here the attacker only saves the input to provided digests, not to every possible input. So, brute-forcing is a volatile approach to reverse hashes, which needs to be redone for every other run.

The most notable two tools to brute-force different hash algorithms are: Hashcat [14] and JTR [15]. Further, we deploy our brute-force framework on different setups for both tools, most notably on our High Performance Cluster (HPC) of the University of Würzburg.

5.1. Architecture

In this section we discuss our general approach in order to brute-force hash digests. We show our work-flow and architecture surrounding current software, which so-called *cracks* a hash digest. Cracking in this context means, that we revert a hash digest back to its input. This is exactly our problem of hash reversal.

We provide our general architecture in Figure 5.1. The central part of hash reversal is done by a *Cracking Unit*. We need to provide two parts for that unit. One is the basic space on which to operate. This is provided by our *Mask Generation Unit*. The other is to include an actual request to reverse hash digests. Here, we provide an *Hash Reversal Unit*.

Cracking Unit. Generally, such cracking software is used to brute-force different hash algorithms. As brute-forcing a hash function is considered a rather computational expensive task, cracking software depends on its user to limit the computational overhead. This is normally done by limiting the to-be-searched input space.

The input space in this context is called *charset*. Charsets are defined space of specific characters. One prominent charset is for example all lower-case letters with a length of

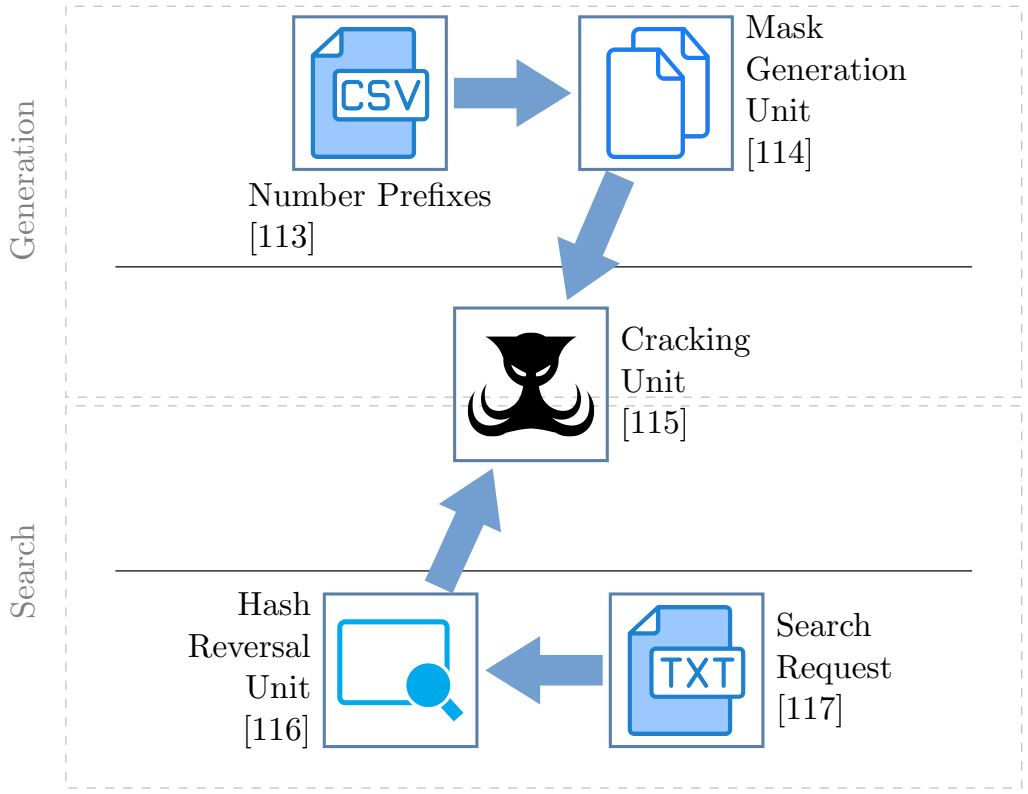


Figure 5.1.: Architecture

maximum six letters. Those charsets can be defined by basically all available characters in ASCII or Unicode – depending on the targeted software. In our application, the charset is defined by the definition of phone numbers. So, we have at maximum 15 digits with a leading plus sign – according to Section 2.1.

To generate those charsets, the cracking software enables the user to input some concrete knowledge of the targeted hash digest. Either the user has some information about the input space of the digest or the user is just guessing some limiting factors on the charset. In those cases the cracking software allows the user to define for example complete word-lists, masked inputs or letting the software generate some charset.

In our scenario word-lists don't make sense, because they would entail all 118 billion numbers. This would be limited by space again. For the other generation methods, most of them are based on the notion of real-world passwords. In those scenarios, the attacker tries to crack an entire password dump. The attacker would iterate through the entire dump multiple times, thereby using already cracked passwords to infer information about the other digests. Similar approach is not possible with the hash reversal of phone numbers. We already know the entire possible input space. Therefore, we don't need to infer more information about the dump by analyzing the already cracked digests. This leaves us with an option of using so-called masked attacks and hybrid attacks.

Masked Attacks. With masked attacks on hash algorithms the user can define prefixes and suffixes and append them with wildcards for other specific charsets. In our case this charset are all digits. Here, we can provide our number prefixes with the plus sign and append them according to our findings in Section 2.1.

Hybrid Attacks. Another attack form we consider with brute-force attacks, are so-called hybrid attacks. The hybrid attacks fuse masked attack and word-lists. The number prefixes are classified by their variable length and then put into a corresponding word-list. So, each

mask length has its own word-list. We execute every mask length with their corresponding word-list in a single command.

In contrast to other hash reversal methods like databases in Chapter 6 and rainbow tables in Section 2.5, brute-forcing software generally does not save results of its calculations. For example, if we calculate a digest, which is not being searched, that result is discarded. So, if we re-run the cracking software and now search for this digest, we need to traverse the entire character space again to reverse this specific digest.

Further, the underlying cracking software doesn't explicitly save the results. The software outputs the reversed digests, however, its internal state only keeps track of results to some extent. For example, when a user tries to reverse a hash, that was already done, the cracking software can directly reverse the hash using an internal database. While this is a convenient feature, we do not rely on this and perform fresh searches.

Another important factor of the cracking software is the support of the targeted hash function – and to what extent. We focus on the SHA-family, more specifically SHA1, because mobile messenger are basing their anonymization on SHA1. As SHA1 is an older algorithm and is also frequently used to hash passwords, this specific algorithm is broadly supported. Nonetheless, to achieve the optimum reversal rate, we also require the cracking software to utilize the GPU to crack SHA1 hash digests.

Mask Generation Unit. This unit preprocesses our mobile phone number data-set. The set contains all possible prefixes of phone numbers spanning all countries. Nonetheless, we need to process the set in order to be able to include them into our *Cracking Unit*. The applied restrictions on the phone number set are described in Section 2.1.

While the set includes the prefixes and their possible length range, we need to map this to the concept of mask attacks. We generally don't define masks as a range, but as prefixes with a specific number of wildcards. So we divide the ranges into a set of possible number length. All numbers with the same amount of wildcards are then merged into a cumulative list – each as a new line.

Once the lists are created, the corresponding number of wildcards are added to the end of each line. Thereby creating our mask-lists, which we can use for the *Cracking Unit*. Those files are purely text-based.

This is similarly to our hybrid attack. There we put prefixes with the same mask-length into separated files. Those files are then used in combination with its specific mask. The specific mask and corresponding file are then used together in a single command. In contrast to masked attacks, we need to run the *Cracking Unit* for each mask-length individually.

This process needs to be done only once. The results can be used generally for every hash reversal process using the architecture depicted in Figure 5.1. However, these lists define our search space inside the *Cracking Unit*. They are also responsible for the duration of the hash reversal and the eventual success rate.

Hash Reversal Unit. In order to reverse the hash digests back to phone numbers, we also need to process possible *Search Requests*. The requests contain multiple hash digests, that need to be reversed.

For the *Cracking Unit* we need to put all digests into a text-file, split by a line break. This text-file can then be directly used as an input for the *Cracking Unit*.

The *Search Requests* themselves are generated by us. They contain values randomly chosen from all possible mobile phone numbers eligible for messenger services, such as WhatsApp. The random algorithm we use hinges on the supposition, that all mobile phone numbers

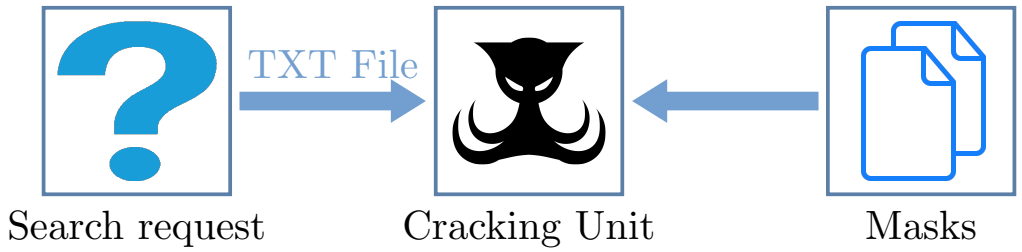


Figure 5.2.: Flow of request

are equally distributed. It is not clear – and probably not true –, that providers distribute the numbers equally on their prefix-spectrum. Nonetheless, we evaluate the viability of this hash reversal approach. For that, an uniformly distributed number spectrum suffices. However, the hash reversal could be improved, if the accurate distribution is known.

Flow. We depict the general flow of a possible search request depicted in Figure 5.2. We suppose, that a breach of some kind already has happened – according to our attacker model in Section 3.9.

So, with the hash digest at her disposal, an attacker can query our architecture to reverse those digests. First the attacker processes the dump to accommodate the *Cracking Unit*. Once the text-file is finished, the attacker can launch the *Cracking Unit* and wait for the results.

As we already mentioned above, the underlying number space needs only to be processed once and can just be reused for every Search Request.

5.2. Instantiation of Architecture using Hashcat

In this section we detail our application of Hashcat using the architecture depicted previously in Figure 5.1. For that we adapt the different units to our needs. For the *Mask Generation Unit* we discuss the implementations for *mask attack* and *hybrid attack*.

Cracking Unit. We instantiate the cracking unit of the architecture using Hashcat. The reasons we use Hashcat are due to its current development, Graphics Processing Unit (GPU) acceleration and apparent widespread use.

To use Hashcat we need to introduce some initial settings. First, we must define our hash function to be cracked. In our case this is SHA1 according to the use of it in Signal. Hashcat offers several variations applying SHA1. Mostly those variations circle around the notion of including salts and boxing different functions. However, we focus on the simple version of SHA1. Hashcat codes this in a Hash-Mode with the number *100* [118]. Those modes are included using the flag *m* using Hashcat's Command Line Interface (CLI).

Another part of using Hashcat is to choose the form of attack. Hashcat offers nine different attack modes. The basic ones are word-lists, combination attack and brute-force attack. The modes that interest us are the brute-force attack and a combination attack. We choose them, because we want to test on-the-fly hash reversal applying Hashcat. In Hashcat brute-force attacks are called and defined as mask attacks. The combination attack is called a hybrid attack and fuses mask attacks with word-lists.

As Hashcat makes heavy use of the GPU, it offers the option to define its load on the underlying system. The different load profiles are called *workload profiles*. Hashcat defines four different types:

Number	Performance	Desktop Impact
1	Low	Minimal
2	Default	Noticeable
3	High	Unresponsive
4	Nightmare	Headless

Table 5.1.: Hashcat Workload Profiles

Our setups normally run in headless environments. As such, we make use of the fourth option called "nightmare". This allows us to use our GPUs to their maximum. We can define this option with the flag *w* using Hashcat's CLI.

Mask Generation Unit with Mask Attack. This unit prepares the use of so-called mask attack. We here adapt this concept concretely to Hashcat. Mask attacks are "but more specific" than classic brute-force attacks, because we can define per position a specific charset. As such, mask attacks let the attacker define the input space more detailed. As described before, our input space is generally composed from numbers, except for the leading plus sign.

In Section 2.1 we describe specific prefixes for phone numbers. Those prefixes are sorted by length of the variable task. Hashcat allows us to map this variable part with the placeholder *?d* per position in the phone number.

Now, Hashcat doesn't just use masks, but allows us to define a file containing different masks – separated by a new line. Those files have the file extension called *.hcmask*. We call our file in the following command examples *mask.hcmask*.

We show an excerpt of an example file in Listing 5.1. We can see, that while candidates have the same prefix, we define every possible length as its own entry. For example, the prefixes *+49169* and *+49168* are represented with three, four and five positions of digit variation.

In order to show the process of the *?d* wildcard, we exemplified the procedure in 5.2 based on *+49169?d?d?d*. This placeholder results in 1000 different numbers, from which a hash digest is calculated. Most notable, this procedure of mask variation is done directly on the GPU.

```
...
+49169?d?d?d
+49168?d?d?d
+49164?d?d?d
...
+49169?d?d?d?d
+49168?d?d?d?d
+49164?d?d?d?d
...
+49169?d?d?d?d?d
+49168?d?d?d?d?d
...
```

Listing 5.1: Hcmask-File

```
+49169000
+49169001
+49169002
```

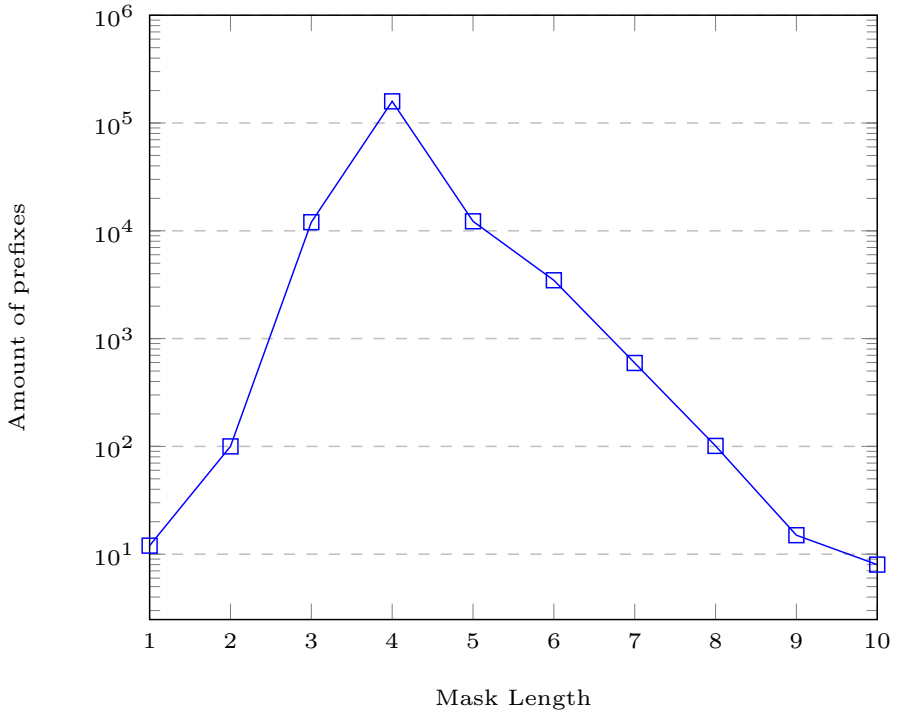


Figure 5.3.: Number of entries per mask length

```
+49169003
+49169004
...
```

Listing 5.2: Hcmask example results

As the mask length has a performance impact, we want to show the general distribution of the different mask lengths in Figure 5.3. Here we can see, that the distribution centers around a mask length of four. In fact, this mask length represents around 84.76% of the included prefixes. The accumulation of the different mask lengths results in a *.hcmask*-file with 187483 entries.

Mask Generation Unit with Hybrid Attack. In contrast to the previous mask attack, we generate here multiple files. The files contain only the prefixes of a specific mask length. Because we have only ten different mask length through the *Number Prefixes*, we generate ten files containing all mobile phone number prefixes.

The mask itself is used in the Hashcat command. As we use here ten different files with different masks, we also need to execute the *Cracking Unit* ten times on the same Search Request. We generally don't execute those ten commands concurrently, because the GPU is already on high load with a single execution.

Hash Reversal Unit. The task of this unit is relatively simple – we need to put a given Search Request into a format understood by Hashcat. The request is comprised of hash digests computed from phone numbers. We put all digests into one file. The different values are separated by a new-line. This file is inserted via Hashcat's CLI.

More important to our tests is the way we generate the example search requests. We apply a uniform distribution over all possible phone numbers. For that, we index all possible numbers and map each prefix to its respective span of indexes. For randomness, we use the system-provided random number generator.

5.3. Instantiation of Architecture using JTR

In this section we apply the generic architecture with JTR. We describe here the details of the different Units of the Figure 5.1. So, we first describe our *Cracking Unit*, *Mask Generation Unit* and *Hash Reversal Unit*.

Cracking Unit. The *Cracking Unit* centers around the hash cracking software JTR. We choose this software, because its commonly referred to as an alternative to Hashcat. It offers also GPU acceleration, SHA1 computation and is designed to use brute-force.

With this instantiation we only focus on the hybrid attack. Mostly because there is no equivalent to the *.hcmask* files with JTR. For this attack, we define four arguments for JTR. The first one is our word-list. For the second argument we provide the corresponding mask. The next input is our format to-be-cracked. The fourth input is a list of to-be-cracked hash digests of mobile phone numbers.

We put this all together in one execution of JTR for each mask length. The generic command is shown in Listing 5.3. The command *john* is used to execute JTR. This command depends on the method of your installation. Further, the *w* flag includes the prefixes with *prefixes_10.txt*. The mask here targets a ten digit variation of the phone number prefixes. The *?d* defines the digit range from 0 to 9. The *?w* denominates, where the word (i.e. phone number prefix) is included. For the format we choose the fastest one for SHA1 digests. The last argument with *10k_digests.txt* includes our *Search Request*.

```
john -w=./prefixes_10.txt -mask='?w?d?d?d?d?d?d?d?d?d' \
--format=raw-sha1-opencl ./10k_digests.txt
```

Listing 5.3: JTR exectuion example

Mask Generation Unit. For this unit, we split all phone number prefixes into mask-length and include them each in a simple *txt* file. In the Listing 5.3 this is done with the *prefixes_10.txt* file. We also provide the other nine prefixes with the corresponding mask-lengths.

Hash Reversal Unit. This unit is similar to our instantiation with Hashcat. We put the *Search Request* into a single line-separated txt-file. In the Listing 5.3 we use the file *10k_digests.txt* for this purpose. This file doesn't change for all ten executions of *jtr*, because the file includes all the digests, that need to be reversed.

6. Hash Database

In this chapter, we present our approach for inverting hash digests of phone numbers using databases. We first discuss our general architecture in Section 6.1. This architecture is then instantiated in Section 6.2 using Redis and in Section 6.3 using LMDB – in those sections, we fully detail our implementation.

6.1. Architecture

In this section, we discuss a general approach on how to generate and how to store number and digest pairs. Inverting hash functions can be done for a specific charset by saving every possible input within that defined space with its hash digest in a database. In our case the charset would be a phone number and a 160-bit SHA1 digest. Naturally, a multitude of types of databases exist, most notably relational databases such as MySQL [34].

Every type of database engine could eventually be used in our use case, but if we look closely most databases are not designed for this. MySQL (or any Structured Query Language (SQL)-based database), for instance, is a relational database. Their main focus lies on putting tables into relations with each other and allowing the user to navigate and join them.

But for inverting hashes we just need to store two values. As a so-called `key` we store the resulting hash digest. The key is to be stored in such a way, that we can do a look-up by the hash digests. We also need to save a so-called `value`, this is the phone number generating the key digest.

We split our approach into two sides. One side is tasked with generating the look-up table we use as our database. The other side is used to handle search requests for the database. We provide an architectural overview of our hash reversal approach based on databases in Figure 6.1.

6.1.1. Components

The architecture in Figure 6.1 includes following components: Number Prefixes, look-up Table Generation Unit, Phone Number Database, Hash Reversal Unit and Search Request. We discuss and explain their details in this section.

Number Prefixes. As we already discussed in Section 2.1, phone numbers consist of a prefix and a variable part provided by the specific provider of the prefix. We also narrow

these prefixes down to those, which are accepted by common service providers, such as Google, Facebook or WhatsApp. We call this compressed list of prefixes our *Number Prefixes*. This list is used as an input and basis for our *Generation Unit*.

Look-Up Table Generation Unit. This unit handles several tasks itself, but the main goal is to generate the *Phone Number Database*. The unit has the *Number Prefixes* as an input and filters those entries based on the targeted number prefix range of the hash reversal.

The unit then processes the prefixes and generates phone numbers from this input. The numbers are then hashed and further processed as pairs of phone numbers and their hash digests.

We parse pairs into a form, which can be inserted into the database using its Application Programming Interface (API). We then insert all calculated key/value pairs. Once the database is populated, this unit's task is finished and we can use our *Phone Number Database*.

Phone Number Database. The database component is based on a regular key-value database and stores key-values generated by the Look-Up Table Generation Unit. We don't alter the database engines in any other way than to tune the provided configuration options. The database is accessed via their officially supported API and provides our architecture with a central place to store the computed pairs.

In our case the database is like a bridge in this hash reversal process, as depicted in Figure 6.1. It enables us to connect the two different components for generating the input and searching the data with a real-world tested access protocol. In fact, the enhancements provided by the database are directly linked to the performance of our approach. So, the usage of different databases will result in different performance, which is exactly what we evaluate in this chapter.

Hash Reversal Unit. This unit's task is to provide an interface to bulk-search for hash digest based on phone numbers. To achieve this task, we provide two elements inside the component. For one, we allow a search request in a text-based form to be inserted via a Text-based User Interface (TUI). For the other, we preprocess the included hash digests in order to search their reversed values in the *Phone Number Database*.

Search Request. In most of our tests, the Search Request is an ordered list of hash digests provided by an user. The digests are searched in the database and thereby reversed back to a phone number. Please note, that we are the user in this thesis and provide the request ourselves.

Flow. In addition to the architecture, we divide the flow of our process into three steps. First, we need to create the key/value pairs according to a phone number and its digest. Second, we need to save the resulting data in some form. In our case we choose two different Databases with different features. And third, we need to create an interface to lookup those values in a fast and concise manner.

In order to give an overview, we provide Figure 6.1. Here one can see the flow from the initial number prefixes, which are predefined, to an intermediate step as key/value pairs. Those values are then stored in our database. Once the database is filled, a user can issue a query to reverse a given hash digest back to a phone number.

Two of the more popular key-value databases are Redis database and LMDB. We choose these two, as both are currently developed, while one uses RAM for speed-up and the other is designed to be persistent data storage. We discuss both their instantiations in following sections.

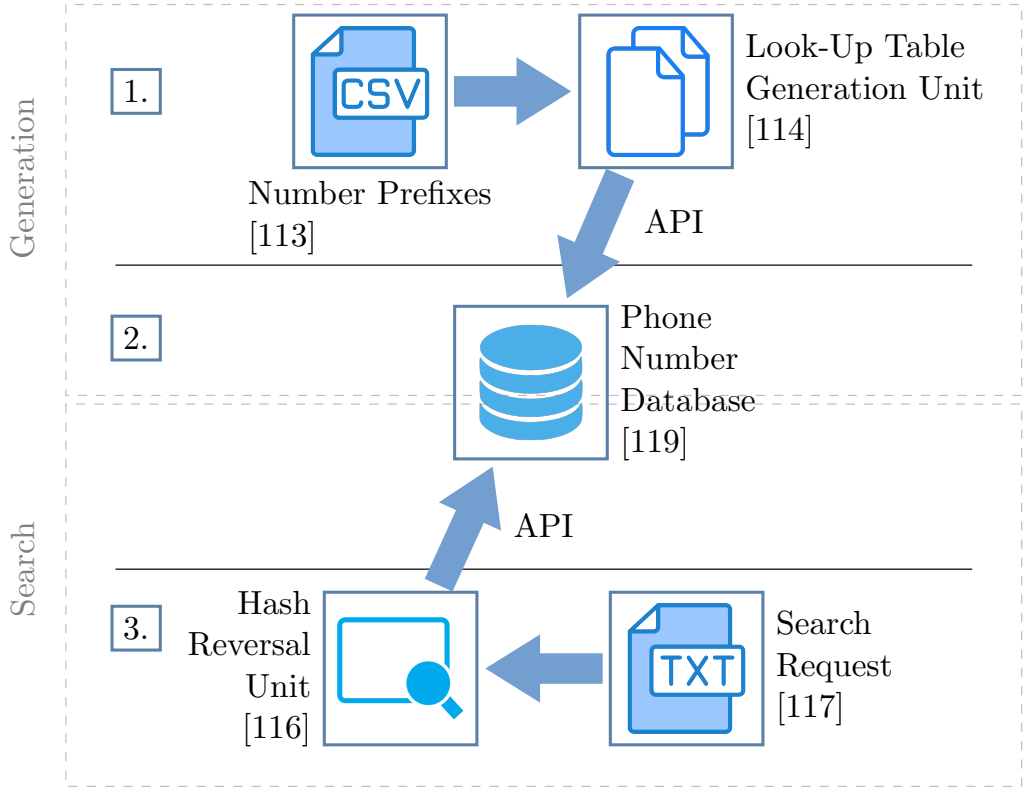


Figure 6.1.: Architecture

6.2. Instantiation with Redis

Redis is considered one of the most performant databases currently in existence. Its uses range from disk cache (i.e. like memcached [120]) to production databases. Even in trendy application like BigBlueButton [121], Redis acts as a center piece to facilitate high performance.

Our decision to use Redis is based on its key-value nature, praised performance and open-sourced code base. Most central are the following features Redis provides. It is designed to only exist in RAM and therefore can implement a specific data structure called hash tables. This promises solid performance for our application. We explain the inner workings of Redis more detailed in Section 2.3.

For our use case of reversing hash digests Redis offers sufficient aspects to implement our general architecture. For instance, the key-value nature is perfect for storing the hash and its preimage. The promised constant lookup-times should also allow for an instant reversal in an on-the-fly setting.

In this section, we further discuss the instantiation and implementation of the generic architecture of Section 6.1. We describe our implementation of all components depicted in Figure 6.1 in the following sections for Redis.

6.2.1. Implementation of Look-Up Table Generation Unit

In this section, we describe our implementation of the Look-Up Table Generation Unit. The authors of [2] provided a Comma Separated Values (CSV) conforming to their findings of valid phone numbers. We use those values as a basis for our testing samples. The phone numbers are filtered by a Python script to include the formatted data into a header file – as a hard-coded part of our tool-chain. We depict an overview of the unit in Figure 6.2.

We first describe our handling of phone numbers with their representation, filtering and finally inclusion for our program. Then, we detail the generation of numbers and the consequent hash digest calculation. Last, we describe the API used for insertion.

Representation of phone numbers. Due to the extra characters like + as discussed in Section 2.1, phone numbers are normally saved as a string. As such, we are also using a string representations for phone numbers.

This is not an optimal solution in terms of space. For example the number 9 needs to be saved as an ASCII form $0x39$ instead of just $0x9$ in hexadecimal. This results in a generally doubled storage overhead. Hence, it might seem reasonable, to build a transformation tool from string representation to a decimal representation for internal storage to save space. This can be further researched in future works.

The string representation of phone numbers is obviously important for the calculation of the hash digest. The binary form of both the string and decimal representation are different and as such result in different hashes. Thus, we represent phone numbers as string in our tests for all steps.

Filter. We generally try to use the entire number prefix space to populate the database in our tests, but one can filter the input based on country codes.

Once the filter is set, we preprocess the prefixes in such a way, that the program can further process the input. The *Number Prefixes* only contain the prefixes and their supposed length. We unify these two characteristics by appending the prefixes with the variable length as Xs. For example, a prefix like *4993100* with one variable digit is formed into *4993100X*. Each X then stands for the digit-range *0-9*.

Data structure in Header. The data is held by a multi-dimensional char-array. Typically, char-arrays are defined by the maximum length of each string – in our case 18 chars – and the total number of entries. The number of entries is dependent on the country which numbers are to be reversed. In a world-wide deployment, this would mean all possible phone numbers.

Generation and Calculation of Key/Value pairs. We then deploy a multi-threaded generation tool to get the hash values and phone numbers. This tool is written in *C* and uses the libraries *glibc*, *OpenSSL* and *pthread*. It includes the previously mentioned header file with all to-be-processed phone number prefixes.

We choose *glibc*, because it is the standard library for *C* in *gcc*. *OpenSSL* and *pthread* are chosen, because of their convenience. Both tools provide easy and direct interface with their functionalities. For instance, one possible alternative to *pthread* is *threads* library. However, for our case, *POSIX* threads suffice and are per default supported by our setup.

The unit first iterates through our char-array containing the prefixes. For every number in the char-array a thread is created. As Linux systems have a maximum number on threads, we implement a maximum number of concurrent threads allowed in the program.

Within each thread, a recursive function traverses the phone string and replaces every X with a zero. Once the function is deep enough – i.e. all Xs are replaced – the hash digest is calculated and added to an output file.

After that, each position is iterated from zero to nine and hashed by our tool. The results are saved for each thread separately into a simple text file. We first encountered some problems with this approach, as there is also a limit on open file descriptors in Linux. This problem was solved by limiting the concurrent thread count.

Insertion of Key/Value pairs. We deploy Redis with volatile memory, therefore we loose all added data once the database resets. This introduces an unnecessary bottleneck for our

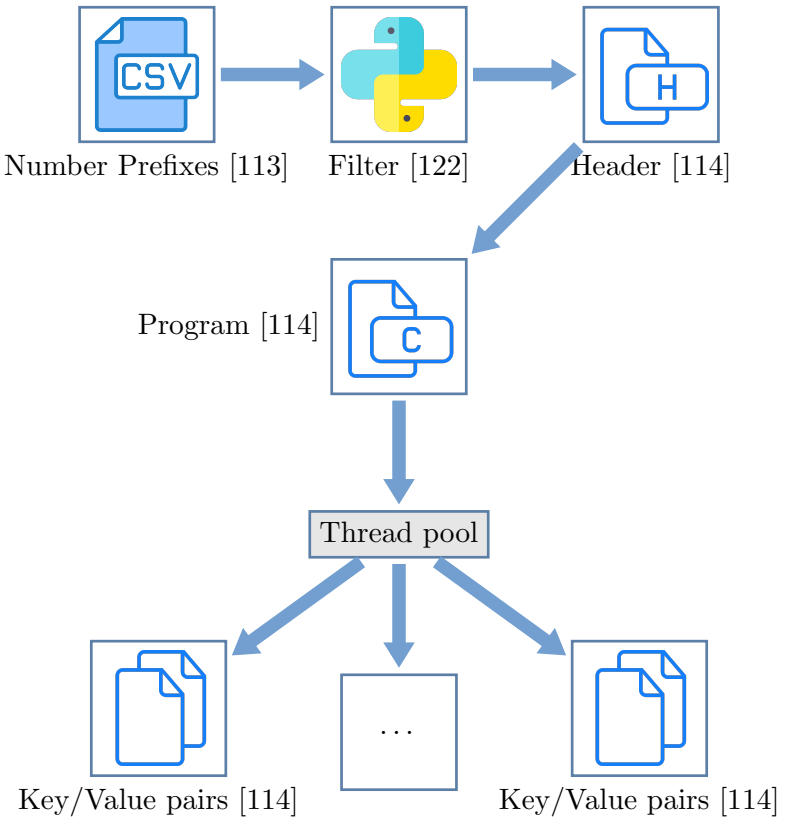


Figure 6.2.: Flowgraph of generation unit

testing. So, we initially calculate a sample of phone number digests according to Section 2.1 and save them intermediately onto a Hard Disk Drive (HDD). This intermediate step limits amounts of hash calculation and therefore allows a faster testing cycle.

We can insert the text files containing key/value pairs with *redis-cli*. This command-line interface for Redis allows us to pipe the calculated pairs from the HDD into our database. We perform a so-called mass insertion in accordance to Redis' best practices [123].

In later iterations of the insertion process, we directly insert the pairs into Redis with the *C*-library *HIREDIS-VIP* [124]. Without the intermediate storage of key-value pairs on the HDD, we achieve faster insertion times. We depict the process in Figure 6.3.

6.2.2. Implementation of Phone Number Database

We show in this section our configuration of the Redis database engine. In our first setup, we run a single instance of Redis. Our initial optimizations revolve around disabling the persistence schemes of Redis. Per default, Redis creates so-called snapshots of the database. As we discussed in Section 2.3, we don't need this feature and therefore disable it to avoid a bottleneck.

RAM allocation. Redis generally allocates as much memory as the underlying system offers. This creates some problems. Once the database reaches the maximum amount of available space, the system could worsen performance due to swapping or block other services on the server to run smoothly. In a worst case scenario, the system hangs itself.

To avoid these problems, we simply set a maximum amount of memory for the entire instance according to our setup. For instance, one setup allows us to manage 612 GB of RAM. To guarantee the correct execution of the rest of the system, the maximum amount of memory Redis can allocate is capped to 600 GB of RAM.

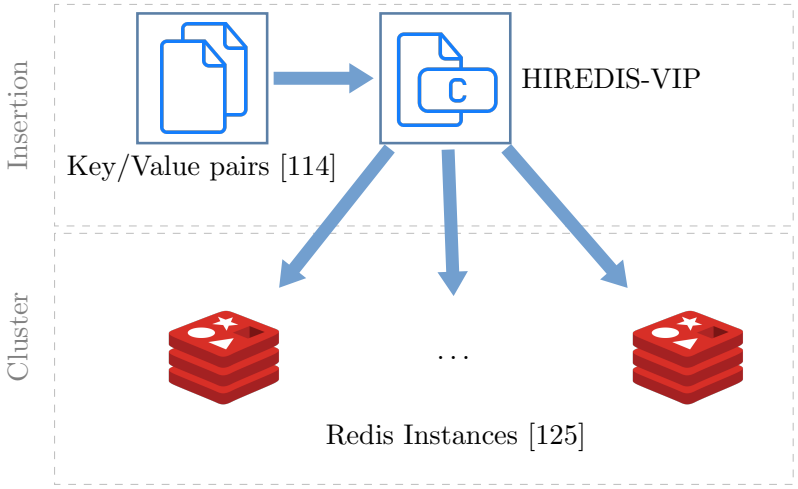


Figure 6.3.: Flowgraph of populating Redis cluster

Eviction Strategies. With only a maximum memory available to store data, Redis offers eight different eviction strategies once the memory is exhausted and new pairs need to be inserted. The strategies revolve around three possible options:

- Expiration date of the data
- Timestamp of last-used data
- Frequency of usage

In our use-case, none of those features are important. For one, we don't need to set an expiration date for the data. Once a phone number and hash digest pair is generated, the pair needs to be available for the lifetime of the database.

For the other, neither frequency nor last-used attribute are essential for our task. When and how often the data is accessed, is not important, because we want to reverse all possible numbers.

Thus, we use a no-eviction strategy. If we recap our goal here, we want to show the feasibility of such a database and its rough resource consumption. So, there is no need to evict one pair for another, as there is no difference in those pairs for our use-case.

In other words, once data is included in the database, it remains for the duration of the database. Another side-effect of choosing this eviction strategy is its improved performance, as we don't need to delete evicted data and insert new data.

Cluster Choice. Theoretically Redis should be able to handle 2^{32} keys. The Redis FAQ [126] even states, that a single instance can handle around 250 million keys. Unfortunately, we can't reproduce this in our use-case. In the first test-runs our Redis instance crashed multiple times with roughly over ten million entries.

At first, this is an unexpected behavior. After some investigating, we traced down the issue to segmentation faults. This circumstance prompted us to make use of the cluster functionality of Redis.

Redis Cluster. A Redis cluster running on the same host has a rather simple architecture. The basis are concurrently run Redis instances, each with its own configuration – we just start the program a lot of times.

To combine the cluster, Redis offers an interface, in which the instances are treated as buckets. As discussed in Section 2.3, Redis uses a CRC16 error-correcting code on the keys and divides the results into 16384 slots with a modulo operation.

Each running Redis instance is then linked to one or more of those slots. So, once the cluster-CLI receives a request, this request is redirected to the corresponding instance based on the key.

We use the provided script for cluster creation from Redis GitHub page [127] and change it for our needs. The script itself allows us to define the number of instances, their respective configurations and takes care of log- and lock-files [128]. We included our exact parameters in Appendix A.

Cluster Configuration. In our case, we use 120 instances to handle our tests. In order to have a functioning system, we split the available RAM equally for the instances. For instance, with a setup with 600 GB RAM each instance has a maximum memory of 5 GB. With this configuration, we are able to fill the entire RAM of our test setups.

Cluster API. We use the C client *HIREDIS-VIP* [124], which supports clusters. First, we create a standalone program, in order to input the raw data of a file provided by our Generation Unit. In later iterations – now with a stable Redis – we merge the two tools into a test-suite to create and insert phone number and digest pairs directly into the running Redis cluster.

One note on the implementation with *HIREDIS-VIP*. The library is compiled and installed based on their GitHub instructions. Nonetheless, the project is a little bit outdated – with its last current entry around 3 years ago. Per default, the examples provided by *HIREDIS-VIP* are not functioning with current versions of Redis. This issue was solved by using the function *redisClusterSetOptionRouteUseSlots()* in accordance to [129], which sets *HIREDIS-VIP* to use Redis' cluster slots.

6.2.3. Implementation of Hash Reversal Unit

In this section we describe our implementation of the Hash Reversal Unit. We implement this component as a Python script, which uses Redis's cluster API called *redis-py-cluster* from Python's internal package manager *pip*. The library enables us to discover every single instance of Redis without the need to provide a direct link to every instance. We then use a simple loop and the provided *get* function to access and reverse the hashes.

The Search Request is provided in a text file, which lists all digests separated by a newline. We then transform the provided digests into an API-conform form. The search itself is done sequentially in the order of the digests.

In our test, we are able to fill the entire 600 GB RAM of our Redis cluster with 3.8 billion entries. This is around 3.2% of the entire search space of 118 billion entries. In fact with our current optimizations, an attacker would need around 19 TB of RAM to fit in all entries.

While this amount of investment is currently only possible for an attacker with a larger wallet, the gain is substantial. Our test indicates a constant lookup time for multiple batch-sizes of 0.1 ms, allowing an attacker to reverse hash digests of phone numbers in real time.

Even with the need to scale up the environment, Redis makes this easy due to their concepts of clusters. An attacker can simply deploy 30 times more instances on different machines to have instant look-up times. However, communication overhead needs to be investigated in future work.

6.3. Instantiation with LMDB

In this section, we present our instantiation of the general architecture as described in Section 6.1 with LMDB database. Additional background information on LMDB database and B+-trees is provided in Section 2.4.

The LMDB is a widespread database, that is most prominently used within SQLightning [130], Postfix [131] and even Monero [132]. It is generally shown to have great performance in comparison to other databases. There are three key properties of LMDB database that make it well suitable for the targeted application:

- Key/Value database
- B+-tree
- Memory-mapped file

Key-value approach. One reason is its key/value approach. We focus on the reversal of hashes, so we only need to store two values per entry. Some more complex approaches like relational databases are too bloated and would generate unnecessary overhead for our use case.

B+-tree data structure. The B+-tree data structure is interesting to have as a contrast to Redis. In comparison, Redis is designed to be in memory exclusively, which allows them to implement other data structures, such as hash tables. Instead LMDB is designed to be also persistent on a hard disk. In fact, the performance of a database is slowed down by access times of the underlying hardware. Redis' data is only residing in RAM, that has constant random access times. However, LMDB is targeting HDD as the location for the data. This should mean worse access times for random instead of sequential access.

Memory-mapped file. The other reason to choose LMDB is its memory-mapped file. As we show with Redis, an in-memory database has some advantages, such as lookup-times. LMDB allows its users to have the advantages of persistent storage and an in-memory database during its run-time. This allows us to have a better comparison to Redis as both are in-memory databases – if the data completely fits into the RAM.

6.3.1. Implementation of Look-Up Table Generation Unit

In this section, we describe our implementation of the Look-Up Table Generation Unit centering around LMDB. The unit is written in C. It takes the phone numbers – filtered by a python script – as an input in form of a string array. This array is within its header-file and is generated from the initial phone number prefixes. We detail this unit in Figure 6.4.

Filter. The python script filter the *Number Prefixes*, as the entire database is considered too big for our test environments. We enable the filter-selection by different country codes – i.e. "US" for the USA or "DE" for Germany. The amount of different numbers is currently limited by our test-bed.

Data structure in Header. The header is identical to header file in the instantiation of Redis in Section 6.2.1.

Generation and Calculation of Key/Value pairs. The unit makes use of the *OpenSSL* library – similarly to instantiation with Redis – in order to calculate the hash-digest of the generated phone numbers. Obviously, *OpenSSL* is not the only library that can be used to achieve this task. However, *OpenSSL* is the default library of our underlying OS Debian and as such is chosen by us for simplicity reasons. The unit obviously also uses the development library for LMDB. The library is provided by the APT package manager with *liblmdb-dev*. To use the library we set the flag *llmdb* in *gcc*.

Initiate LMDB. Once the key/value pairs are calculated, the unit makes use of the system-provided LMDB library to initiate and access the database. In order to initiate the

database, we need to provide some initial settings. This step is not needed in the instantiation with Redis, because Redis already runs in the background.

For one, we define a preliminary size of the database, as the resizing of a B+-tree is resource costly. This is done with the function *mdb_env_set_mapsize()*. We set the maximum size of the database to 500 GB for all our test. We considered only the maximum size of RAM for this value. One can also create larger databases, but this will result in performance penalties, once the data needs to be loaded into the RAM.

For the other we can define the maximum number of readers of the database. In our instance, we define only one reader, as we only access this database from one client. We use the library-provided function *mdb_env_set_maxreaders()* for this.

The user can also set some additional environment flags, such as no file sync after a commit or making the database read only. While those flags can be used to affect the behavior of the database, we use none of the possible flags. In our initial search for some performance gains, no flag stood out to us to achieve better performance.

Insertion of Key/Value pairs. After the initial setup is completed, we can commit our request by opening a so-called transaction. Within one transaction, we can provide several inserts into the database. However, those inserts are only committed to the database, once the transaction itself is committed. Another aspect to keep in mind is, that dangling transactions can result in performance degradation. This is due to the immutability of the database within every open transaction, forcing the system to keep track of the different versions.

For the insert command *mdb_put*, one can also pass flags to change some nuances of its behavior. Our unit inserts key/value pairs of digests and phone numbers. Thus, we insert the hash digest as the key for an entry. By providing the flag *MDB_NOOVERWRITE*, we make sure to not insert the same values twice. The scenario in which the key of two different values is the same is be negligible, because this would be a collision inside the hash function.

When the unit is finished with the generation of the database, the database and environment access are closed. The resulting file is our Phone Number Database.

6.3.2. Implementation of Phone Number Database

In contrast to Redis, LMDB has no background service or daemon to actually serve the data. With LMDB the phone number database is represented by a single file, which can be accessed like any other file. This also means, the database is fundamentally handled by the OS.

Clearly, most other databases also rely on the OS to handle basic hardware interactions. While a file can be accessed via a file descriptor, LMDB makes use of so-called memory-mapped files. In this case, the OS maps the file directly into the virtual memory of the running program. Consequently, the program has faster access times than handling the descriptor with read/write system-calls, but relies more on the OS with the write-backs to disk.

The database can be opened by multiple programs concurrently. In fact, once the database is generated, it could be shared and copied to multiple other instances as a B+-Tree on disk. This is opposite to the volatile configuration of Redis, which just can save transactions, but not the internal state of its hash tables.

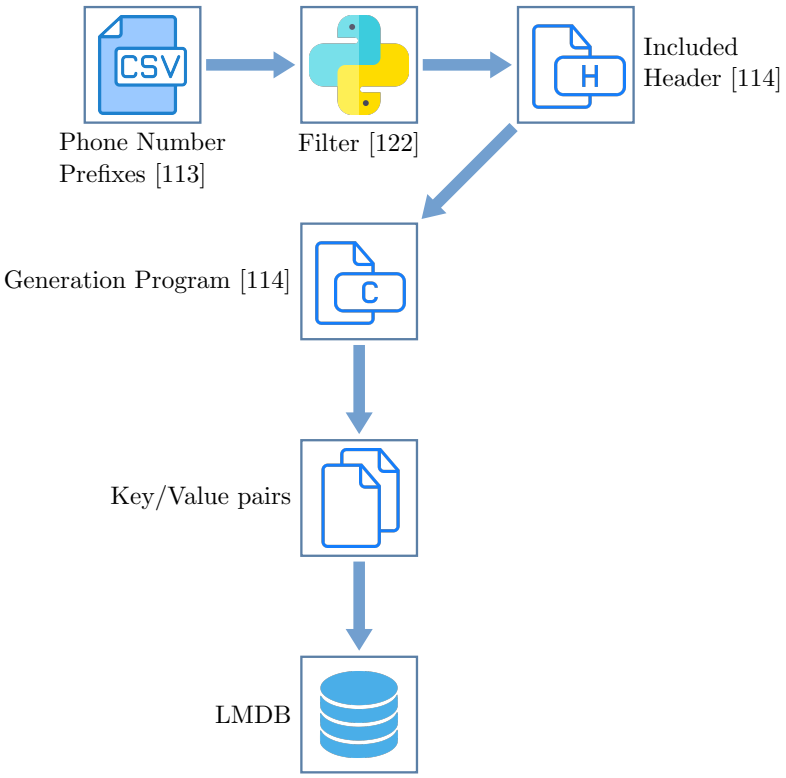


Figure 6.4.: Flowgraph of LMDB generation

6.3.3. Implementation of Hash Reversal Unit

Similarly to Redis, the queries to LMDB-based hash reversal database are also managed by a C program. We use the same API of the Look-Up Table Generation Unit in Section 6.3.1. We just make some adjustments, since we only want to read the database, instead of filling and writing to it.

We see in Figure 6.5 the work-flow of our Hash Reversal Unit. Once the attacker has a dump of hash digests, she can formulate a Search Request. The request is then compiled into our search program using a generated header. The search program then contacts the database. If the digest is found, the attacker gets the reverse of the hash digest – back to phone numbers – as a result.

Initiate LMDB. One of those adjustments is setting the database as a read-only memory-mapped file. We can do this by setting the flag *MDB_RDONLY*, once we start the database environment. We further set *MDB_NOTLS*. This flag allows us to handle locks on the database. As such, we can have multiple concurrent read transactions in our thread without possible performance penalties.

Search program. After the environment is set, the database file is read into the memory by the OS. As soon as the file is mapped, we can access the database with the function *mdb_get()*. This function allows us to get a single value with a key. As a side note: We can only access a single value of one key. In some other use cases – where multiple values are associated with a single key – we would need to use LMDB's cursors.

As the "get"-function allows us to only retrieve data with one key, we need to consider some alternatives to access data in bulk. For one, we could access one key per transaction. Obviously this would result in unnecessary overhead by opening and closing a transaction every time. Instead, we use a single transaction to access all requested data in bulk. The values for the keys are then printed by the *printf()* function for testing purposes.

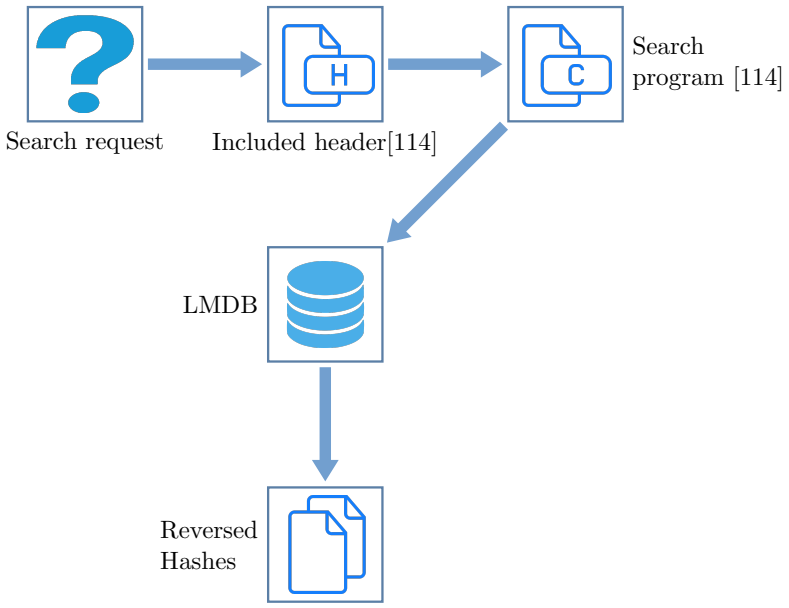


Figure 6.5.: Flowgraph of LMDB hash reversal

Search Request and Reversed Hashes. The search values are included via a header file. We then iterate through them within our single transaction. As explained above, we use the get-function *mdb_get()* next, in order to actually search the database. If the function can't find a key, the search aborts. We want to measure the lookup-times and size of the database for hash digests. Thus, we don't want to reverse a realistic data dump and to calculate false-positive rates and therefore implement an abort.

7. Evaluation

In this chapter we provide rigorous evaluation of three different hash reversal techniques in the context of Contact Discovery. We first describe five different test-beds we used in our evaluation, in Section 7.1. Then, we detail our findings of brute-forcing hashes in Section 7.2. Next, the evaluation results for the reversal technique using databases are described in Section 7.3. The third technique, rainbow tables, is evaluated in Section 7.4.

7.1. Test Setups

We describe here several setups that were used in evaluation. All setups are chosen to fulfill the specific needs of our different hash reversal techniques. For instance, hash reversal based on brute-force relies on GPUs and in-memory databases depend on sufficient amount of RAM present.

7.1.1. Test Setup – OpenStack

We name our first setup *OpenStack* and it is intended for Redis' evaluation. We choose this setup, because Redis needs a substantial amount of RAM to store our hash database.

Hardware. The system runs on a Virtual Machine (VM) provisioned by the High Performance Computing Cluster of the University of Würzburg [133].

The cluster is based on OpenStack. Users can provision their VM according to so-called `Flavors` – predefined settings for the VM. In our case, we use 48 vCPUs of Intel Skylake cores at 2.3 GHz with 630 GB of RAM and 1 TB of disk storage.

Software. We choose for our setup the OS Debian 9 with the packaged version of Redis. Generally, the VM is running the standard kernel and has no other optimizations other than discussed in this chapter or done by the HPC. The server is run without any desktop environment and is only accessed via Secure Shell (SSH).

7.1.2. Test Setup – SLURM

Our second setup is named SLURM and it is intended for Hashcat's evaluation. The evaluation is detailed in Section 7.2.2 and focuses on the reversal of the entire mobile phone number space.

Hardware. We use the university's HPC. The cluster offers two dedicated NVIDIA Tesla P100 per node. Further, GPU nodes also consist of two Intel® Xeon Gold 6134 Processor

and 384 GB of RAM. However, we don't utilize the entire node, but share the resources with other deployed tasks.

Software. The HPC is based on Debian 9. In contrast to the other setups, we use here specifically Hashcat in version 5.1.0 and not the version provided by the OS.

The HPC uses the Slurm workload manager [134] to schedule different jobs/tasks. We dispatch our jobs via the scheduler using a bash script.

7.1.3. Test Setup – 1080TI

Our third setup is named 1080TI and it is intended for GPU-heavy tasks. Those tasks are related to our brute-force approach in this thesis. So, this setup is exclusively used for evaluation in Section 7.2.

Hardware. We test our brute-force approach on a Zotac Nvidia 1080 TI. The system is also comprised of an AMD Ryzen 3 3200G and 16 GB DDR4 RAM. However, we specifically only use the 1080 TI in our tests.

Software. We use an at-the-time current *Arch* installation with the up-to-date version of JTR and Hashcat from the packet manager *pacman*. Cuda with *11.0* and Nvidia driver with *450.57* are the most current versions.

7.1.4. Test Setup – Fujitsu

Here, we describe our setup, named Fujitsu, used for the evaluation of LMDB in Section 7.3.2 and of Redis in Section 7.3.1. We choose this setup for our database evaluation, because it offers a sizable amount of RAM.

Hardware. In this setup we use a Fujitsu Primergy TX 300 S8 with 512 GB RAM, two Samsung 850 PRO 250 GB Solid-state Drive (SSD) and four Crucial MX500 500GB SSD. The two Samsung SSDs are operated in RAID0. The four Crucial SSDs are used in a RAID10 ZFS [135]. We build the RAID10 with ZFS by stripping the content across two mirrors containing each two disks. The system also provides 40 vCPUs based on two Intel Xeon E5-2690 v2 at 3.00GHz.

Software. The system runs on Debian 10 "Buster" and is installed on the Samsung SSDs. The tests are executed from the ZFS partition. We make no other modification to the system. This system runs also without any desktop environment and is exclusively accessed via SSH. There are no other optimizations made, neither to the kernel nor to the system.

The LMDB development library is from the official Debian repository. The package is called *liblmdb-dev* and included with *gcc* using the flag *-llmdb*. The version of the LMDB library is *0.9.22* and *gcc*'s version is *8.3.0*.

7.1.5. Test Setup – Manjaro VM

The last setup is named Manjaro VM and it used for the hash reversal using RainbowCrack-NG described in Section 2.6. The software mostly depends on a high Central Processing Unit (CPU) count to achieve the best performance. Thus, we evaluate RainbowCrack-NG with this setup using all available resources. The tool also has a better performance with current software versions, that we use in our evaluations.

Hardware. The underlying hardware is identical to our setup *Fujitsu* described in Section 7.1.4.

Virtualization. We use a virtualized environment with *KVM*. The VM is assigned all 40 vCPUs and is given 32 GB RAM. The image file is on our ZFS partition, in order to have the best available performance.

Software. As our OS, we use Manjaro – an Arch-Linux based derivative. This OS applies a rolling-release update schedule. Thus, we can use the most current version of software – besides compiling it ourselves.

For the hash reversal we use the modified RainbowCrack-NG. The software depends on two system-provided libraries OpenSSL and OpenMP. Manjaro distributes OpenSSL with the version *1.1.1g*. However, OpenMP is a standard included in the gcc. The current standard for OpenMP is 5.0 with *gcc 10.1.0*.

Important note. While the above setup is our setting for the evaluation, we want to stress the importance of a current version of OpenMP, i.e. gcc-toolchain. We initially tried to generate rainbow tables with *gcc 8.3.0*. The compilation and execution worked as expected. However, we manually aborted the generation of one table similar to Section 7.4.2 after 10 hours.

7.2. Evaluation of Brute-Force

In this section we present our findings regarding hash reversal with brute-force attacks. In Section 7.2.1, we compare JTR and Hashcat to show performance differences of both tools. Next, in Section 7.2.2, we use our brute-forcing techniques with Hashcat to reverse phone number digests based on all mobile phone numbers. We repeat the experiment with JTR in Section 7.2.3. Finally we compare our findings in Section 7.2.4.

7.2.1. Comparison JTR and Hashcat

In this section we compare two brute-force tools in the context of reverting SHA1 digests. The hashing algorithm is central, as it needs to be supported by the underlying tools. Generally Hashcat offers a variety of different hashing algorithms, mostly deployable onto the GPU. While JTR also includes numerous algorithms, the tool differentiates the algorithms between CPU and GPU versions.

We benchmark both tools with the target hash algorithm SHA1, since the algorithm is used by messenger apps like Signal. Hashcat doesn't provide a default option to run only on a CPU. So, for Hashcat we can only benchmark a GPU deployment. However, we provide a comparison of CPU and GPU performance with JTR. We additionally benchmark the mask attack on GPU version of JTR, since JTR is the only tool that supports benchmarking for mask attacks. More implementation details and attack modes are explained in Chapter 5.

Notations and Test-bed. Hashcat shows its results in H/s, i.e. Hashes per second. In contrast, JTR uses c/s, which means candidates per second. However, both notations denote the same value.

We use our *1080TI* test setup described in Section 7.1.3. This setup enables us to deploy our cracking software on a current consumer GPU.

Preliminary Tests. We first performed preliminary benchmarks to identify the tool with better perspectives. For that, we use Hashcat's internal benchmark flag *-b* for SHA1 using the command in the Listing 7.1. We then show the benchmark for JTR for one using a CPU in Listing 7.2 and for the other using a GPU in Listing 7.3. Further, JTR offers a specific benchmark for mask attacks, which we measure using the command from Listing 7.4.

```
hashcat -b -m 100
Speed.*.....: 12960.5 MH/s
```

Listing 7.1: Hashcat SHA1 benchmark

```
john --format=raw-sha1 --test
Raw:    21636K c/s real, 21636K c/s virtual
```

Listing 7.2: JTR SHA1 CPU benchmark

```
john --format=raw-sha1-opencl --test
Raw:    64527K c/s real, 64527K c/s virtual, Dev1 util: 15%
```

Listing 7.3: JTR SHA1 GPU benchmark

```
john --format=raw-sha1-opencl --test --mask
Raw:    3166M c/s real, 3150M c/s virtual, Dev1 util: 100%
```

Listing 7.4: JTR SHA1 GPU mask benchmark

Comparing SHA1 performance of JTR on CPU with the GPU accelerated version, the GPU version has around 3x improved performance. However, Hashcat offers a four times better performance than JTR with the mask attack benchmark (cf. Listing 7.4). Using these preliminary benchmarks and a better familiarity with Hashcat, we initially decided to use Hashcat with our HPC of the Rechenzentrum to reverse hashes.

7.2.2. Reversing Entire Mobile Phone Number Space with Hashcat

In this section we present our results on hash reversal of the entire mobile phone number space using Hashcat. We present two attack versions included in Hashcat: Mask Attack and Hybrid Attack.

Mask Attack. We use our prefix database to generate a *.hcmask* file. This file then generates all possible mobile phone numbers. So, with every single run of Hashcat, we need to calculate all possible digests for mobile phone numbers in order to reverse one request.

For the Hybrid attack, we split the prefix database into ten files containing each a specific mask-length prefix.

The mask attack is tested with the *SLURM* setup. The central command in Slurm is shown in Listing 7.6. With that command, we issue the use of a GPU and define our script. In this script, we then use the command *srun* in combination with the Hashcat command shown in Listing 7.5. After we issued a task, we need to wait for a free spot to execute our command. After execution Slurm provides a log-file. We use this file to get execution time and hash-rate of our attack.

```
hashcat -m 100 -a 3 -w 4 question.txt mask.hcmask
```

Listing 7.5: Hashcat command

```
sbatch --gres=gpu:1 -p gpu script.sh
```

Listing 7.6: Slurm command

We firstly define three batches of hashes to be cracked. One batch includes 10k hashes, the second has 100k hashes and the third contains 1000k hashes. We choose those batch sizes to infer performance prediction on a lager scale. We can see the results in Table 7.1.

As seen in Table 7.1, while growing the batch size by a factor of ten, the execution time only marginally increases. This prompted us to evaluate the performance differences of the different charsets in our *hcmask* file. We ran our test for maximum of 30 minutes on every appended mask length from 1 to 10.

Table 7.1.: Batch comparison

Batch size	hashes per second	duration
10k	0.18	15h 18m 28s
100k	1.78	15h 35m 24s
1000k	17.45	15h 55m 2s

Table 7.2.: Mask length comparison

Mask length	kH/s	duration	Group size
1	110	36s	$1.2 \cdot 10^2$
2	999	52s	10^4
3	4242	MAX	$\sim 12 \cdot 10^6$
4	42319	MAX	$\sim 1.5 \cdot 10^9$
5	100000	MAX	$\sim 1.2 \cdot 10^9$
6	104000	18m 2s	$\sim 3.5 \cdot 10^9$
7	104000	3m 57s	$\sim 5.9 \cdot 10^9$
8	105000	2m 12s	$10.1 \cdot 10^9$
9	106000	2m 33s	$15 \cdot 10^9$
10	110000	10m 42s	$8 \cdot 10^9$

With the results of Table 7.2 and the magnitude of numbers in each mask group we can deduct, that our computational time is mostly used on the mask lengths 3-5. Those groups only represent $2.8 \cdot 10^9$ numbers. We attribute this phenomenon to the context-switch in the GPU. Hashcat pushes each mask to the GPU and iterates through all possible variations of a mask. Once as mask is finished, Hashcat pushes the next mask onto the GPU, This accumulates to a substantial performance loss, as every mask has a constant overhead to be deployed on the GPU.

As a potential improvement to be explored in future work, one can move these groups into a database for a faster lookup, while simultaneously using Hashcat to crack the bigger number spaces, thus creating a hybrid between brute-force and database approach of hash reversal.

Figure 7.1 shows dependency of the hash rate from the mask length. The x-axis shows the length of the inserted mask and the y-axis depicts the corresponding hash rate. If we consider the plateau of the hash rate around 110 MHashes/s, we are still around 100x times slower than the initial benchmark for Hashcat on SHA1 with our *SLURM* setup.

In Figure 7.2 we plotted the batch size on the x-axis and the execution time of Hashcat using the entire phone number space on the y-axis. We can see, that the total execution time of Hashcat barely depends on the batch size, but rather on the to-be-crawled space.

In Figure 7.3 we show again the batch size on the x-axis, but this time on the y-axis we plot the normalized time in regard to batch size. Here we can clearly see, that the execution time per single hash digest shrinks with the grow of the batch size. In fact, if we search for 100k phone digests in one batch, we achieve a reversal time of 0.05s.

Hybrid Attack. We use again three batches of different size to be cracked with this attack mode. The sizes of the batches are 10k, 100k, 1M. For our setup, we use the *1080TI* platform (cf. Section 7.1.3).

We run the same experiment as with the mask attack. So, we focus on the general execution

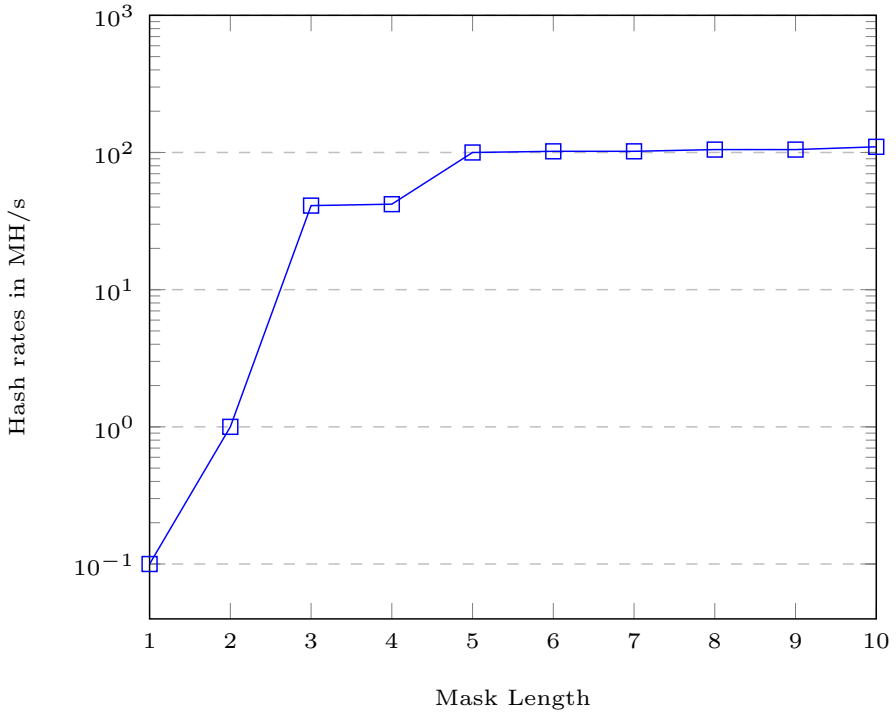


Figure 7.1.: Hashcat Mask Length Mask Attack

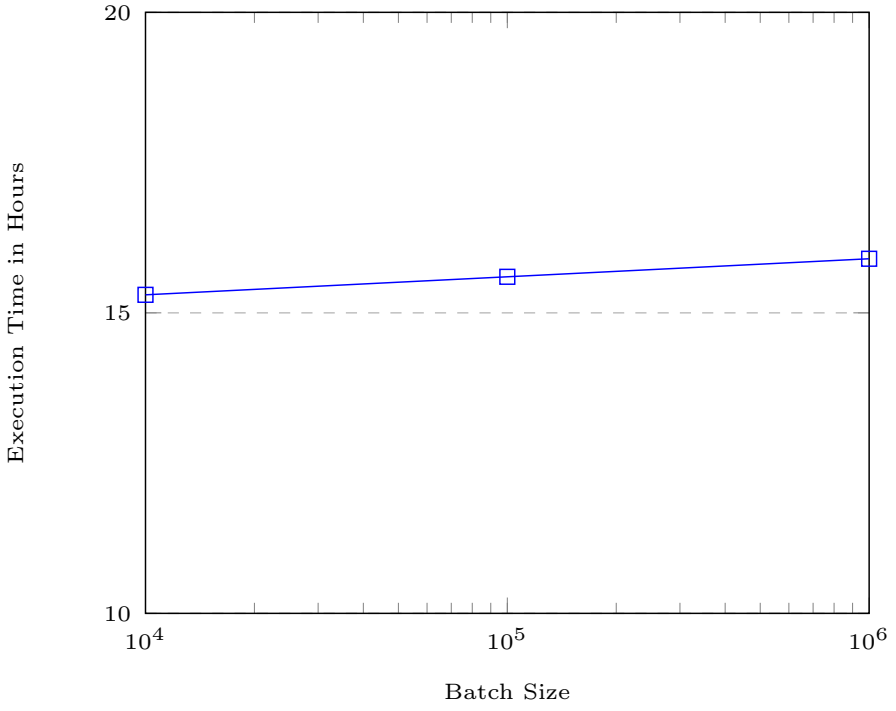


Figure 7.2.: Hashcat Execution Time with Mask Attack

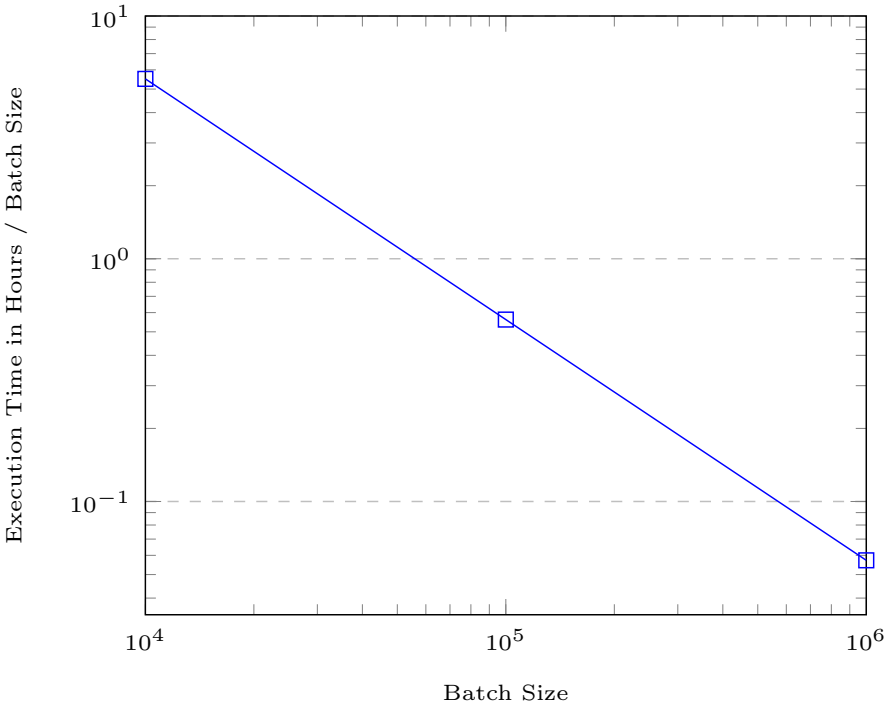


Figure 7.3.: Hashcat Execution Time per Hash Digest with Mask Attack

time, the batch sizes and their cracking rate. However, inspired by the good reversal times with JTR we use a hybrid attack instead of mask attack.

In Figure 7.4 we plot the cracking rate in reference to the mask size. The figure resembles roughly the distribution of number prefixes from Figure 5.3. We further observe a higher cracking rate than previously observed with the mask attack. For instance, the mask length of four produces roughly a ten times faster hash reversal rate than with the mask attack.

Next, we plot the entire execution time of the reversal process in Figure 7.5. With this attack mode, we are able to reduce the execution time of our hash reversal process from roughly 15 hours to 2 hours. In other words, we have an performance increase of roughly 7.5x in contrast to mask attacks.

Finally, we show in Figure 7.6 the normalized hash reversal time per hash digest of the *Search Request*. We see the best performance with the largest batch size. In fact, we can reverse hashes in 8ms per digest with the hybrid attack mode and a reasonably large batch.

7.2.3. Reversing Entire Mobile Phone Number Space with JTR

In this section we discuss hash reversal of all mobile phone numbers with JTR. We use the so-called hybrid attack, which combines mask attacks and word-lists. The procedure is very similar to hybrid attacks with Hashcat. Thus, we focus here on the specifics of JTR, rather than describe general approach. We use the same setup with *1080TI* as in the hybrid attack with Hashcat.

Hybrid Attack. For this attack mode, we plan the identical scenario as the experiment with Hashcat. We first define three batch sizes – namely 10k, 100k and 1M hash digests each. We then use the specific architecture from Section 5.3 with JTR.

Figure 7.7 depicts the hash rate per mask length. We see a similar performance increase to Hashcat with a larger mask length. However, in contrast to Hashcat's hybrid attack, we achieve a constant hash rate around eight GH/s with JTR.

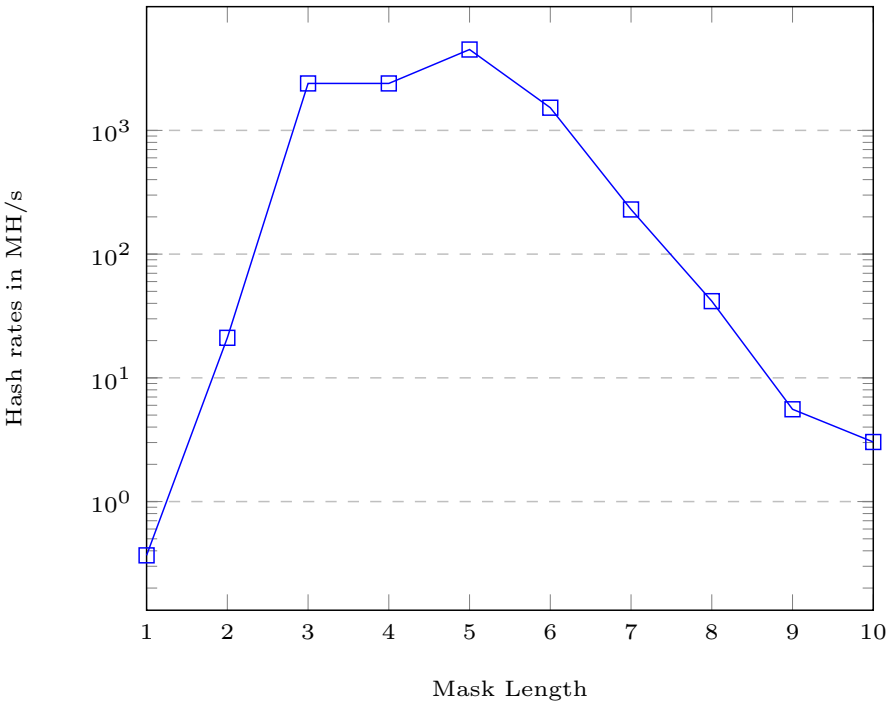


Figure 7.4.: Hashcat Mask Length Hybrid Attack

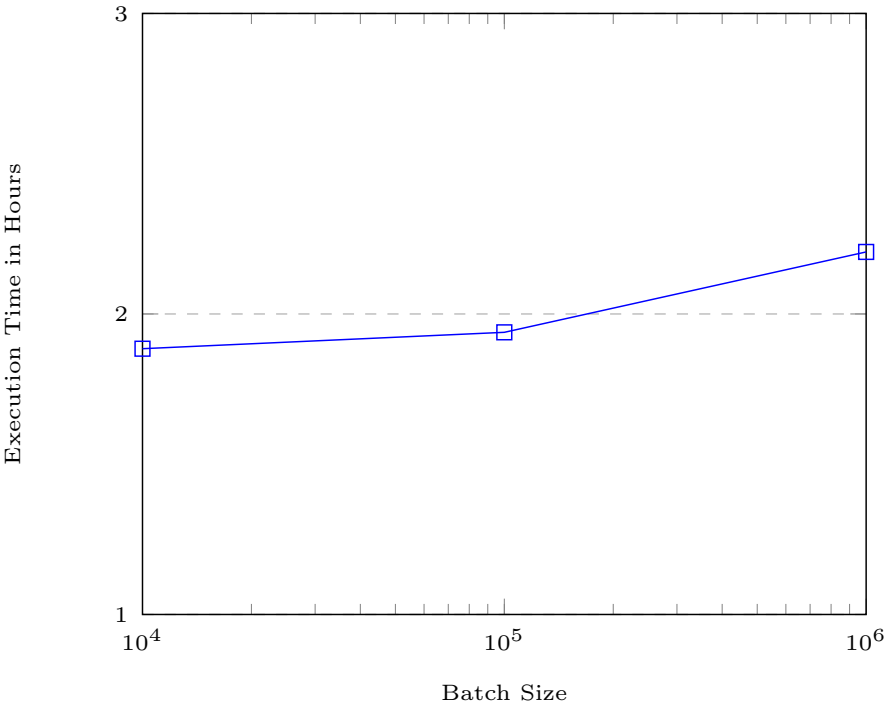


Figure 7.5.: Hashcat Execution Time Hybrid Attack

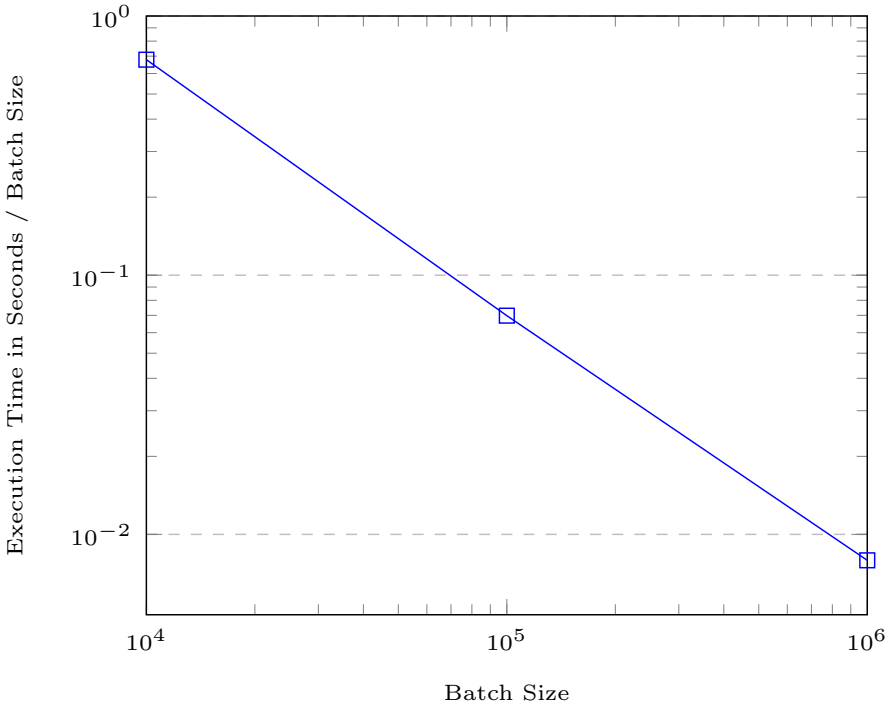


Figure 7.6.: Hashcat Execution Time Hybrid Attack

The total execution time for different batch sizes is depicted in Figure 7.8. Here, we see a vast increase in performance in comparison to Hashcat. While Hashcat's execution times are depicted in hours, JTR achieves execution times under 100s.

We break those time again down, in order to show the performance per hash digest. The Figure 7.9 shows our normalized execution times. For instance, using a batch size of 1M, we achieve a lookup-time from under 0.1ms for a single digest.

7.2.4. Discussion

In this section we discuss our evaluation of the brute-force approach for reversing hash digests. To recap, we first run a mask attack with Hashcat. After that, we used a hybrid attack with Hashcat, which uses word-lists and mask attacks. Finally we run this experiment with JTR against the entire phone number space again. This last experiment shows the most promising results in order to reverse SHA1 hash digests.

We are somewhat surprised to see, that JTR is magnitudes faster than both attacks with Hashcat. It is obviously due to the fact, that JTR utilizes the GPU better than Hashcat. The initial benchmarks showed performance for Hashcat around 12 GH/s and for JTR only 3 GH/s. However, in the real-world tests we have a peak performance of Hashcat with 4 GH/s and JTR with an astonishing 10 GH/s.

Furthermore, the hash rate stays roughly constant with JTR. With Hashcat's hybrid attack we can see a decline in performance with growing mask length. We are not sure, why Hashcat doesn't utilize the hardware the same way JTR does. While one can explore this in future work, we see that the hash reversal times with JTR are already near benchmarking performance. So, even by fine tuning our tools any further, we would see only a marginal gain due to the limitation by our setup.

Now, the performance per hash correlates with the batch sizes of the *Search Request*. We achieved with JTR's hybrid attack a peak hash reversal of under 0.1 ms per hash digest.

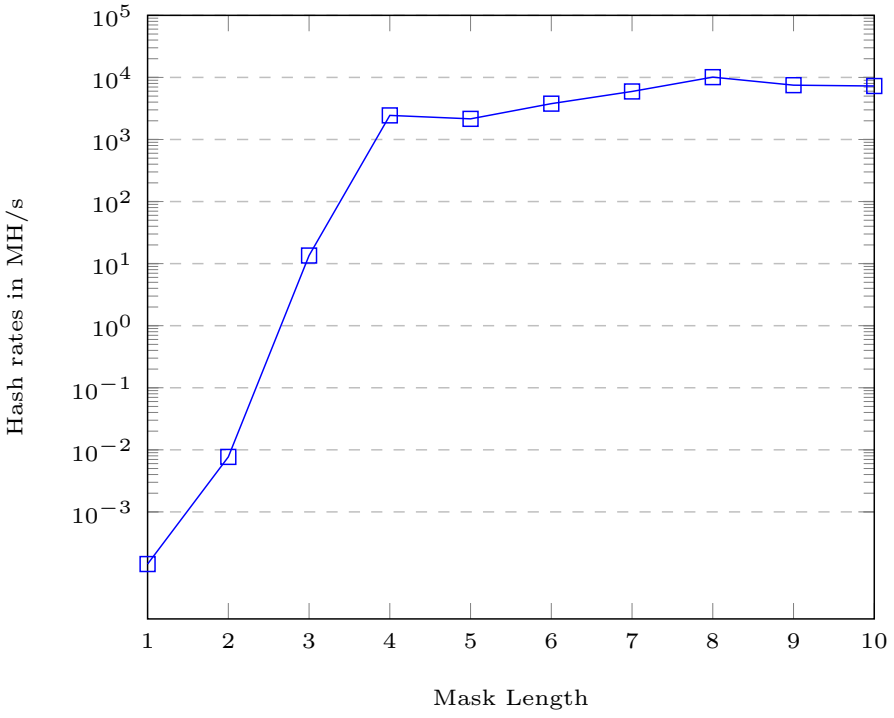


Figure 7.7.: JTR Mask Length

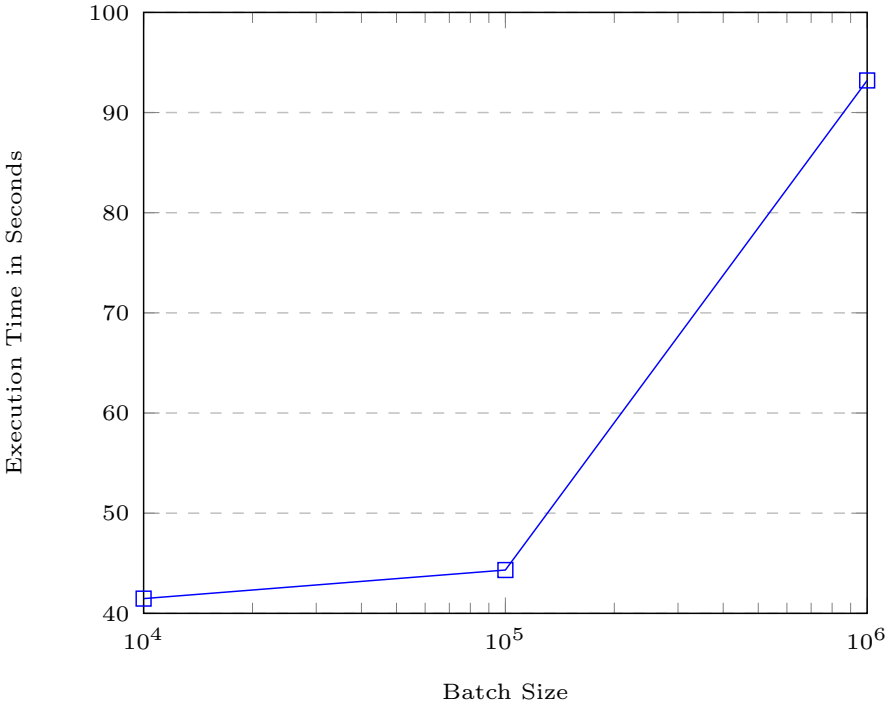


Figure 7.8.: JTR Execution Time

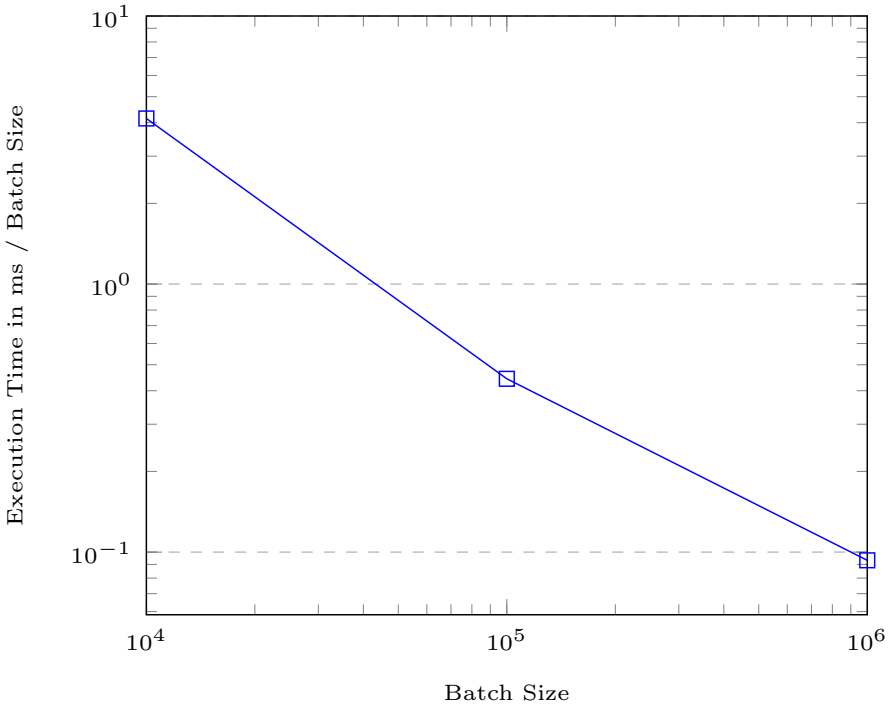


Figure 7.9.: JTR Normalized Execution Time

This is a performance gain of 57000% in comparison to our initial test results reported in [2].

In summary, we can see, that JTR achieves nearly the peak performance provided by the underlying hardware. The hybrid attack allows us to reverse the entire mobile phone number space in less than 100 seconds. As we use in parts consumer grade hardware, we show, that an attacker with a budget of under 1000 € can easily reverse captured SHA1 digests back to mobile phone numbers in a very efficient manner. This also means, that a malicious provider or a states-actor can view a SHA1 digest breakable on-the-fly with our brute-force approach.

7.3. Evaluation of Database

In this section, we report evaluation results the architecture presented in Section 6.1. In particular, Section 7.3.1 is devoted to evaluation of Redis-based system and Section 7.3.2 concerns LMDB-based instantiation. Finally, we compare both approaches in Section 7.3.3.

We consider three features in our tests to compare both databases. The first feature is the growth of the databases. We look at the growth to project how much space is needed to hold the entire phone number space. The second feature is the look-up times of both database engines. The result should be helpful to decide which database to choose, if a fast lookup-time is essential for an on-the-fly reversal of hashes. The third feature is our observed insertion time of data. This one-time aspect is not central to hash reversal, but can factor into the choice between those two.

7.3.1. Evaluation of Redis

In this section we evaluate performance of Redis with two setups. In our first test, we perform measurements with an exhausted memory space (cf. Section 7.3.1.1). The second

test shows Redis' baseline performance (cf. Section 7.3.1.2). For both tests, we use the components defined in our architecture (cf. Figure 6.1). We generate sufficient key/value pairs with our Look-Up Table Generation Unit to fill the entire RAM of our test system.

Look-Up Table Generation Unit. We use all number prefixes defined in Section 2.1 as an input for our Generation Unit. Then we start all instances of Redis in our Redis cluster. We test their general availability with the provided command-line tools.

Once the cluster is up and running, we start our program. First, the number prefixes are taken to generate all potential numbers. Sequentially parts of this number space is used to calculate their hash digests. After the key/value pairs are ready, we use the API to fill the Redis database with them.

Phone Number Database. We use common command-line tools to monitor our cluster. For example, a combination of the tools *free* and *htop* help us to keep track of the system resources, especially regarding the utilization of the system's memory. Once the maximum amount of pairs are inserted and the cluster is running, the database allocates all available memory space. However, Redis generates no other substantial overhead such as CPU usage or I/O disk.

Hash Reversal Unit. We use part of the inserted data as an input for our terminal. The data is processed to be then inserted into our Redis cluster. We use the CLI command *time* to measure the lapsed time per search request. Every measurement is repeated ten times and averaged out.

7.3.1.1. Test 1 with Redis

In this section we test Redis at maximum capacity of the *OpenStack* setup. We generally want to test the three features: Growth, lookup times and insertion time on our databases. However, this test doesn't measure the growth of Redis, because we only measure Redis at maximum capacity.

Insertion Time and Growth. The process of filling the database took around 13 hours and was aborted once all the RAM was used. We are able to fit around 3.8 billion pairs into our cluster using up all 600 GB of RAM.

Lookup Times. We provide an overview of our findings in the Figure 7.10 and Figure 7.11. We measure the lookup-times for 1, 10, 100, 1000 and 10000 keys. The data is plotted on a logarithmic graph to the basis of ten. This is done, because our batch-size grows exponentially instead of linear and this needs to be reflected in the plot of the resulting data.

In Figure 7.10 we depict the batch-size for x-axis and the entire lookup-time for y-axis. We can see here a linear growth in relation of number of search keys and their lookup-time. The data plotted here seem rather smooth, but in fact has some variants due to some background noises generated by the OS and network. The overhead introduced by background noises, however, is negligible compared to lookup time, hence it is not clearly visible on the plot.

The Figure 7.11 shows the normalized lookup-times divided by the batch-sizes. Again, the x-axis reflects the batch-size, but the y-axis now shows lookup-times per one record (and not per bulk). This gives us better insight of Redis' performance regarding single lookup-times. We measure a constant lookup time of 0.1ms per key.

We see smaller variations on the data points. The results show a linear growth of lookup-times in regard to searched keys. This reflects Redis' internal data structure and their guarantee of a constant lookup-time per searched key. We can therefore match the theoretical numbers with our practical evaluation.

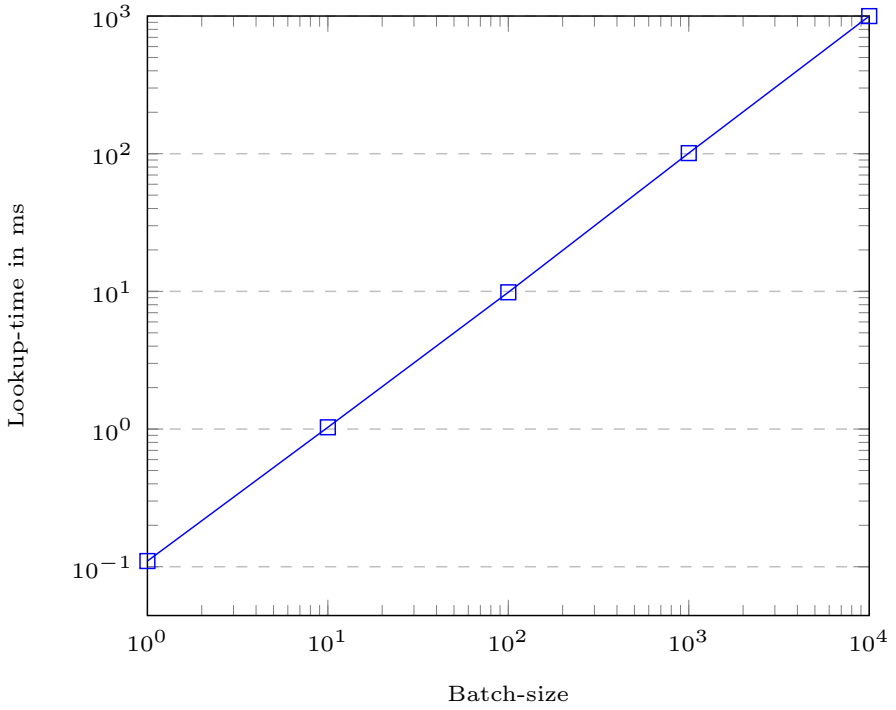


Figure 7.10.: Redis lookup time on maximum Capacity

Summary. Redis seems to be a good choice to guarantee constant lookup-times for an on-the-fly hash reversal setup. The only substantial hindrance is the amount of RAM needed to setup such a database. Renting or buying around 19 TB of needed RAM is manageable and not out-of-the-question, but requires significant costs.

The question arises, if an attacker can rent this amount of memory. One example could be the high memory instances of Amazon EC2, which allows users to rent up to 24 TB of RAM [136]. For instance Those instances are designed to provide a sufficient infrastructure for in-memory database such as Redis or SAP HANA. The costs for an 12 TB instance is 30.54$ per hour for three years [137]. If we look also at the pricing for the 6 TB and 9 TB versions, we assume the cost of 24 TB to be around 60$ per hour. This also entails a 3 year contract, which results in a total cost of around 1.6 million dollars.

7.3.1.2. Test 2 with Redis

This test is based on the *Fujitsu* setup. The Redis version on this test is the rolling release directly from GitHub's master branch. Our API is based on *HIREDIS-VIP* and is also retrieved as its current GitHub version. The goal of this test is not to run Redis at maximum capacity, but instead offer some baseline results in order to provide measurements that can be compared with measurements of LMDB (cf. Section 7.3.3).

Size growth. In Figure 7.12 we see the results of Redis' growth with different amounts of inserted keys. We use the following steps for the pairs included in the database: 1k, 10k, 100k, 1M, 10M and 100M. After the pairs are inserted, we measure the memory usage with Redis' internal variable *used_memory* provided the command *redis-cli info memory*. We make sure, that all keys are inserted by checking the current size of the database with the internal command *dbsize*.

The Figure 7.12 is on both axis on a logarithmic scale. We can see a high initial memory usage of our Redis cluster. This makes sense, because we run multiple instances of Redis. Therefore we generate a high initial overhead, as we essentially run Redis 120 times.

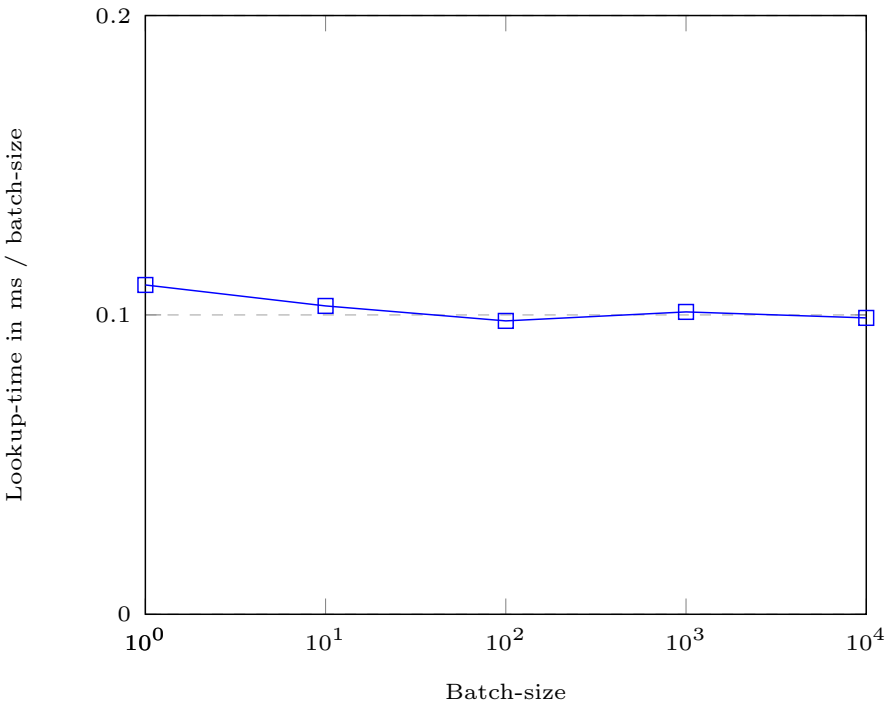


Figure 7.11.: Redis lookup time normalized

Eventually, the size growth approaches a linear growth with more entries inserted – analog with the theoretical complexity of hash tables.

We normalize the growth in regards to the number of inserted values in Figure 7.13. Once the normalization is done, we can see more clearly the growth of memory consumption in regards to inputted values. In fact, we see a decline of per pair memory allocation with growing number of input pairs, because of the high initial overhead. However, the data shown in Figure 7.13 seems to converge to a constant memory growth of 0.2 kB per inserted key.

Lookup times. In this test we further examine lookup times with a fixed amount of inserted records. In particular, we fix the size of the database to 100M entries in Redis cluster in order to see the impact of the Search Request batch-size on Redis. The search requests vary in the batch sizes: 1, 10, 100, 1k, 10k, 100k and 1M. We again take ten measurements and report averaged values.

We see in Figure 7.14 a linear growth of look-up times with growing batch sizes. Thus, we can again confirm a constant lookup-time per single search request query. This implies, an attacker can achieve constant lookup-time no matter how many numbers are included in a search request. Our observations are supported by results plotted in Figure 7.15, where we show normalized lookup-times with the amount of searched keys – resulting in a lookup-time of 0.03 ms per key.

7.3.2. Evaluation of LMDB

In this section we measure the performance of LMDB to reverse hash digests. Here, we use the implementation of our architecture as described in Section 6.3. We first discuss the utilization of the different components in our tests. We run the tests on our *Fujitsu* setup, so we have sufficient RAM for this in-memory database.

Look-Up Table Generation Unit. In contrast to the first test with Redis (cf. Section 7.3.1.1), here we fill the database with a predefined amount of key/value pairs. We use

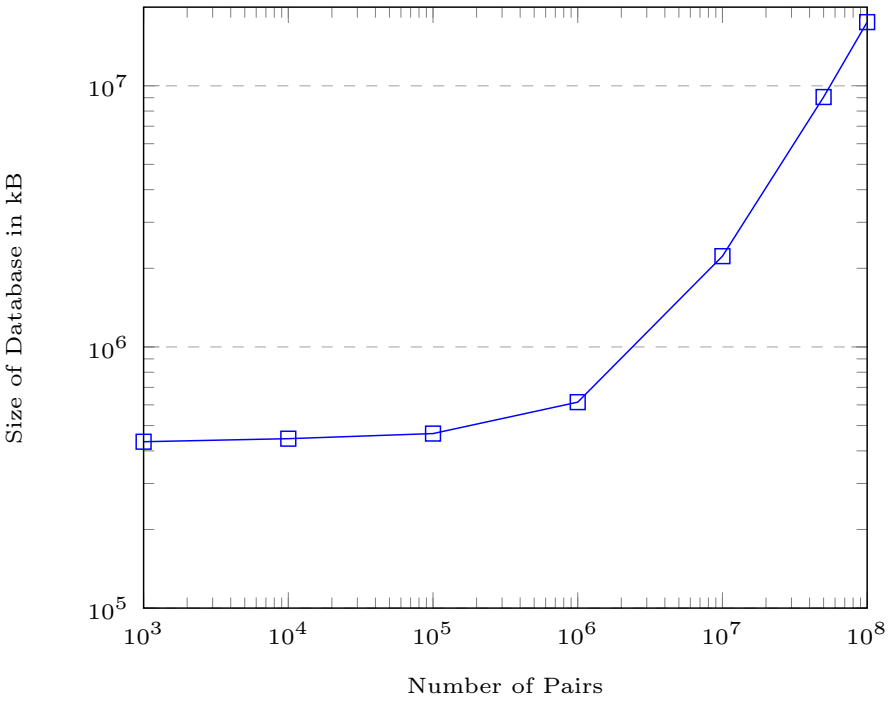


Figure 7.12.: Redis Size Growth

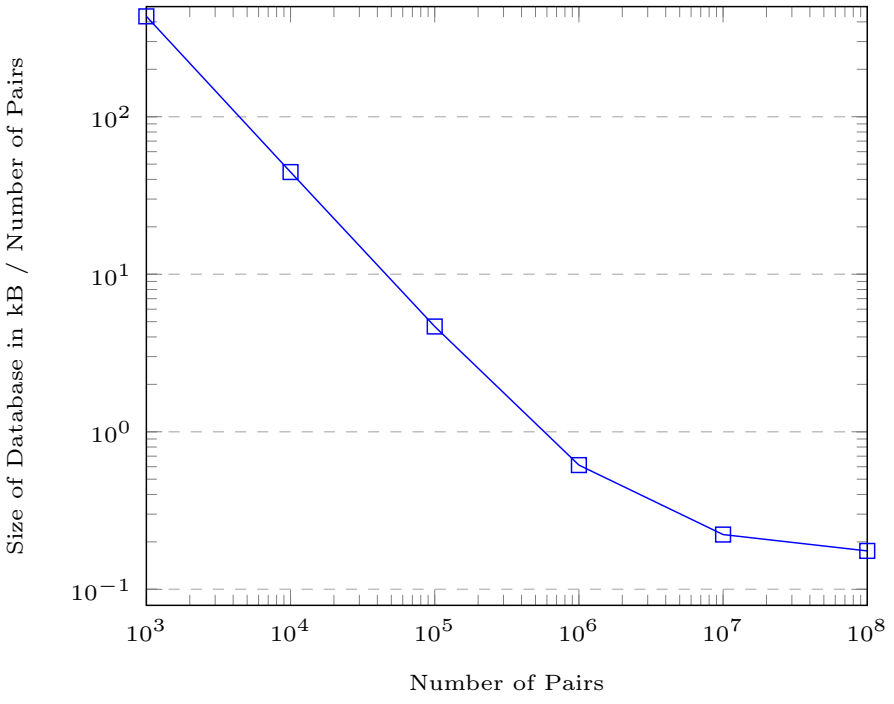


Figure 7.13.: Redis Size Growth Normalized

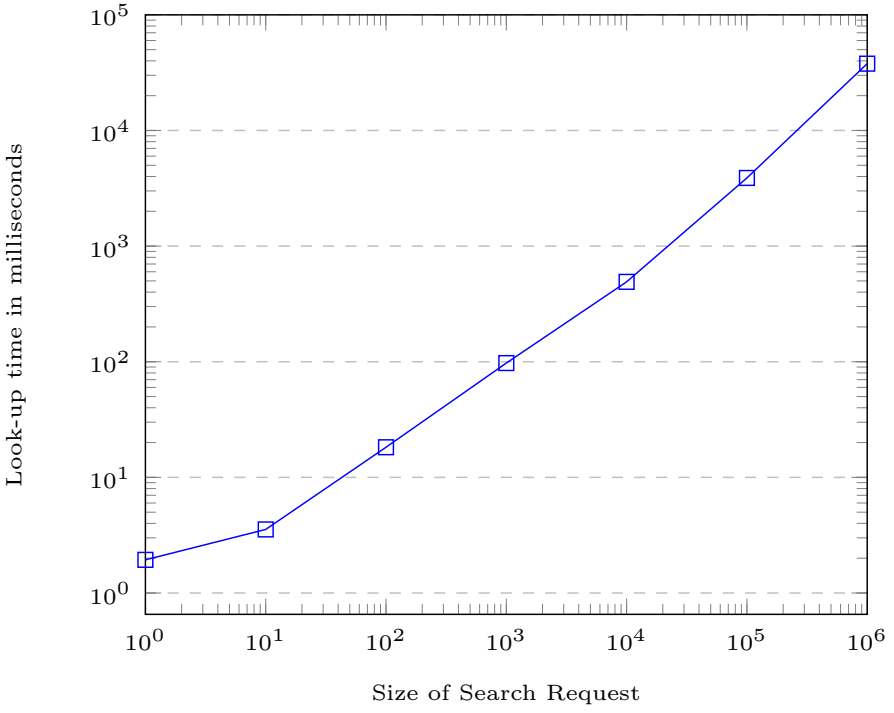


Figure 7.14.: Redis lookup times for different Search Requests

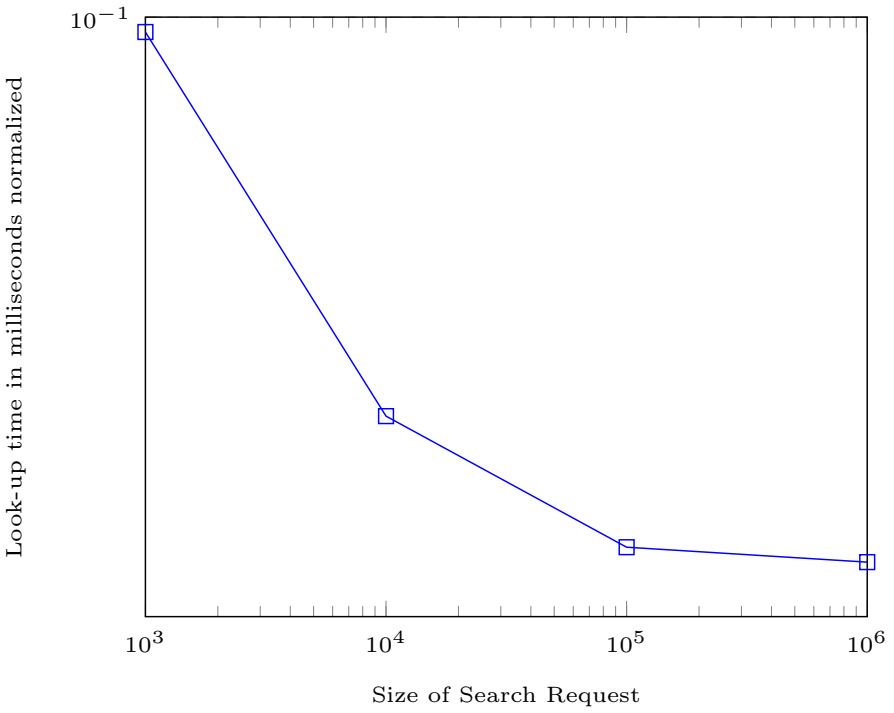


Figure 7.15.: Redis lookup times for different Search Requests normalized

an exponential growth for this amount and use the steps 100k, 1M and 10M. The pairs themselves are randomly chosen, but in an ordered list.

The unit handles this input accordingly to procedure described in Section 6.3.1. We use a one-size-fits-all approach with the size of the database. The size needs to be predefined by us for our tests. We choose the same size of 500 GB for all iterations with 100k, 1M and 10M entries. Please note, that this will not define the size of the resulting database file.

Phone Number Database. The database file is located on the disk. There is no need to monitor the database itself, as this is literally just a file – contrary to the Redis cluster. We show the growth of the finished file in Figure 7.16. We generally use the provided CLI tools to access the database. More specifically, the tools *mdb_dump* are used to see the inserted data and *mdb_stat* to access the size and numbers of the B+-Tree.

Hash Reversal Unit. In accordance to Section 6.3.3, this unit needs to be provided with values to search the database. We use randomly chosen key/value pairs from the database. We assume here, that the database should be complete and therefore every key should be represented in the database. If the database is incomplete, other assumptions need to be reevaluated. For instance, the list of initial phone number prefixes would be then incomplete and needs to be extended with the missing prefixes.

We measure every bulk-search with the *glibc*-function *clock()*. The measurement is repeated ten times and averaged out. As also mentioned in Section 6.3.3, we use one transaction to retrieve all values to the search keys, which commits for each search.

7.3.2.1. Tests with LMDB

In this section we present results of tests of multiple aspects of the utilization of LMDB in context of hash reversal. For one, we examine the growth of the database. For the other, we examine the lookup-time for different amounts of search-request and different sizes of the database. For that, we provide an overview with the raw data in Figures 7.16 and 7.18, in which we use the logarithmic scale.

Size growth. We first look into the size growth of LMDB depending on the amount of inserted values. For that, we fill the Phone Number Database with sufficient key/value pairs. The Generation Unit filters the phone number prefixes according to our predefined number of needed entries. Those are then inserted using the using the approach described in Section 6.3.1. In our case, we create multiple databases with 1k, 10k, 100k, 1M, 10M and 100M entries inserted respectively.

Once the database files are generated, we retrieve their size in kilobytes with the tool *ls*. We can see in the Figure 7.16 a linear growth of database size. The scale is again logarithmic to reconcile the exponential steps in our tests. We also normalize the data by dividing the overall size by the number of inserted entries and show the corresponding results in Figure 7.17. We can see here a constant growth of the database per entry. The beginning of the figure depicts an initial overhead of the database itself. Overall, experiments show, that the usage of LMDB will result in linear growth in regard to the amount of inserted pairs.

This is a positive result, as the overhead of the database structure itself is not substantially bloating the actual size of the inserted pairs. So we can assume, that the added space to the key-value pairs is also at least linear. Thus we can project the size of the database for more inserted pairs.

Lookup times. We then also test the Search Request timings using our Hash Reversal Unit. We use for our tests a search request with 1k entries. The entries are randomly chosen from the inserted data. We run this search on databases with the amount of 1k, 10k, 100k, 1M, 10M and 100M inserted pairs respectively.

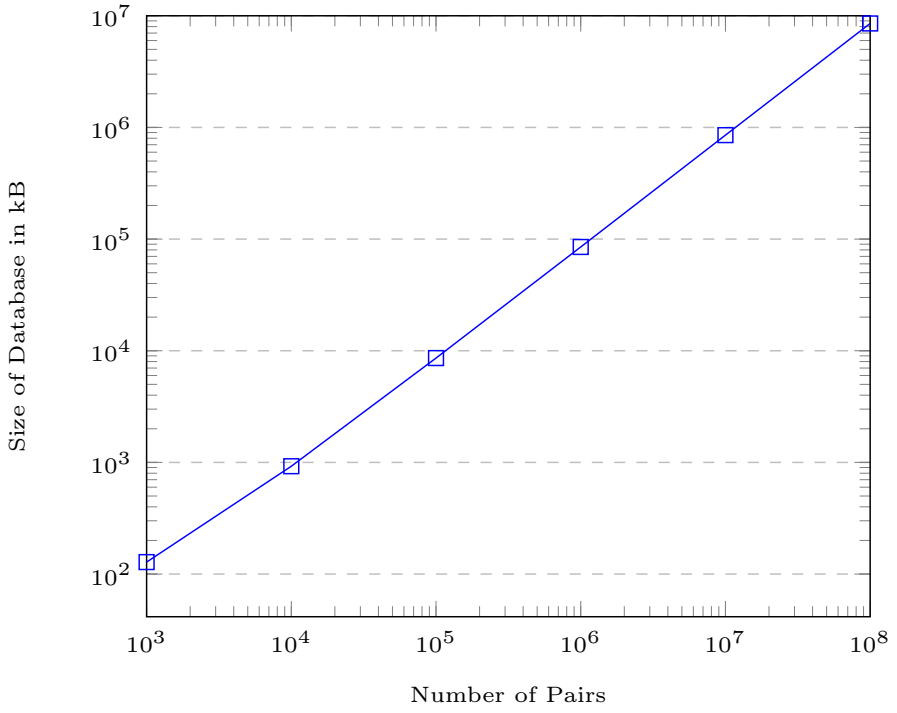


Figure 7.16.: LMDB Size Growth

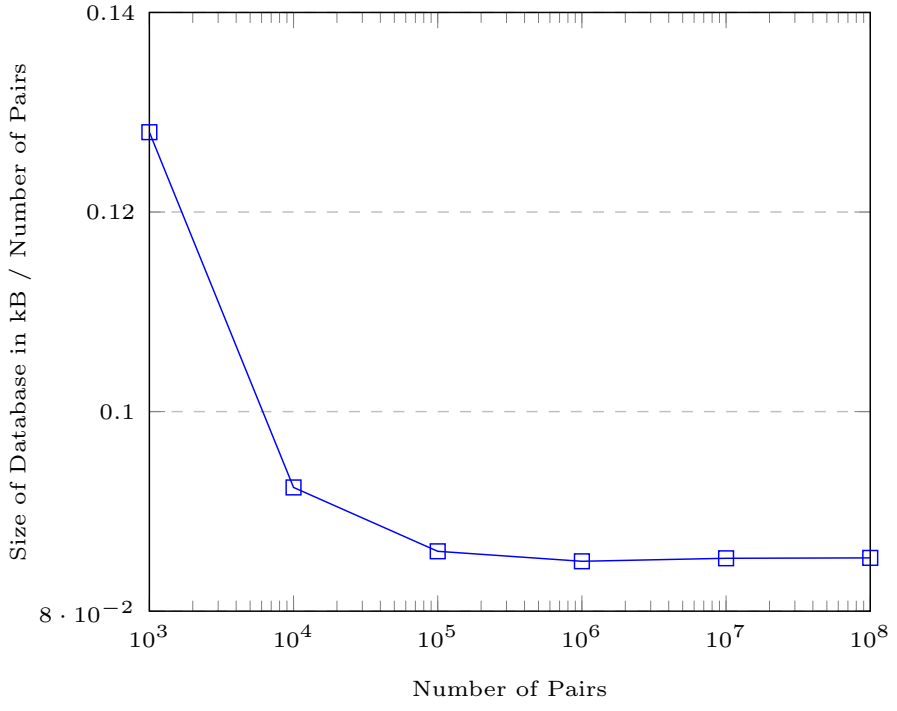


Figure 7.17.: LMDB Size Growth normalized

For this test, we use the C-function *clock()* and the struct *clock_t* from the library *time.h*. We included the measurement of time directly into our Hash Reversal Unit, to have the most direct values. In our preliminary runs, we see, that the printout of the searched-input has a massive performance impact. For example, in the search of the 1k-database and the use of *printf* we see a measurement of around 7ms. However, without this output, the results are around 1ms. Nonetheless, running test without some kind of processing of the data is not typical. Yet, we want here to measure the performance of the database not the implementation of *printf* or the processing time of the results. We therefore measure the lookup times without any output of our unit.

We make sure, that all values are found in our tests, by aborting the test-run if otherwise. As usual, we run every test ten times and average out the results. The tests are measured in milliseconds and depicted in Figure 7.18. Similarly to previous tests, the x-axis is drawn on a logarithmic scale, because our x-values are in exponential steps. The x-axis shows the size of the database in number of entries and the y-axis shows the look-up time in milliseconds.

We see in Figure 7.18 a linear growth in regards to the size of the database. However, we see at the beginning again an initial discrepancy. As we keep the search-size constant on every run, we nonetheless see a growth in look-up times. Most likely this stems from the different depth of the internal tree. Those measurements reflect the fact, that traversing a tree with different depths will result in different times. Furthermore, we observe in our first run of each test bundle initial RAM misses – the first measurement is always the worst. Those misses are obviously due to the first loading of the file into the RAM. The resulting initial overhead is not considered in our numbers, as we want to measure an already running database.

For Figure 7.19 we test a fixed sized database – 100M entries – against differently sized Search Requests. We use the sizes of the requests in steps of 1k, 10k, 100k and 1M values. We see here, that LMDB has linear growth in regards of the inserted pairs. We also normalize the look-up times by dividing with the size of the search request. This is visualized in Figure 7.20. We see here a declining look-up time per searched key. Our empirical measurements confirm the theoretical logarithmic search complexity of the B+-Tree.

All values searched in our previous experiments are sorted. This has some positive effects on LMDB, as the internal cursor can just traverse linearly through the tree. In another experiment, we want to examine this effect. For that, we took 1M random values and search those in a 100M database. We find 2631.5957 ms lookup-time for all searched values. This is around 0.002631 ms per key/value pair.

In our sorted test, we measure 0.001886 ms per key/value pair. So, searching for random pairs increased our times by around 140%. This overhead can be eliminated by preliminary sorting the search request, or the lower performance can just be accepted.

Initiation Time. We observe in our tests, that the initiation time for LMDB degrades exponentially. The Figure 7.21 shows the results the initiation time for different amount of inserted keys. The measured times for 1k, 10k and 100k inserted keys roughly grow linear with around 0.038 ms per key. However, with 1M inserted keys, the insertion time rises to 0.049 ms per key. This trend continues, with 10M having a normalized insertion time of 0.068 ms per key and with 100M keys an insertion time of 0.288 ms per key.

Especially in the test with 100M entries, we see a faster insertion time for the first 50M entries in contrast to the last 50M entries. This may be a problem stemming from its architecture and needs more investigation in future work. As a possible workaround, we could deploy the same clustering mechanism as Redis. The keys could be arranged into

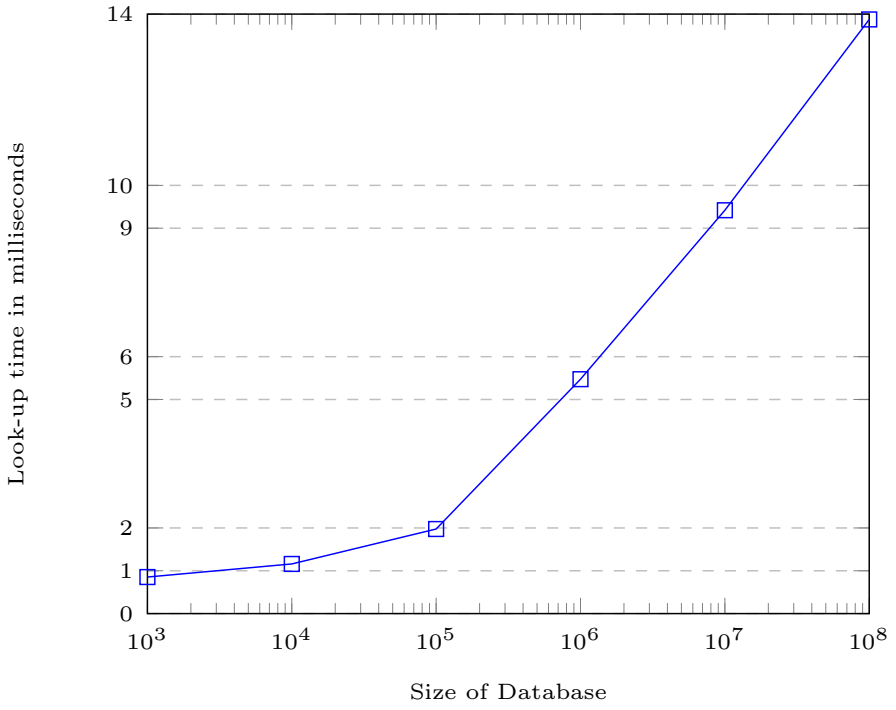


Figure 7.18.: LMDB lookup times

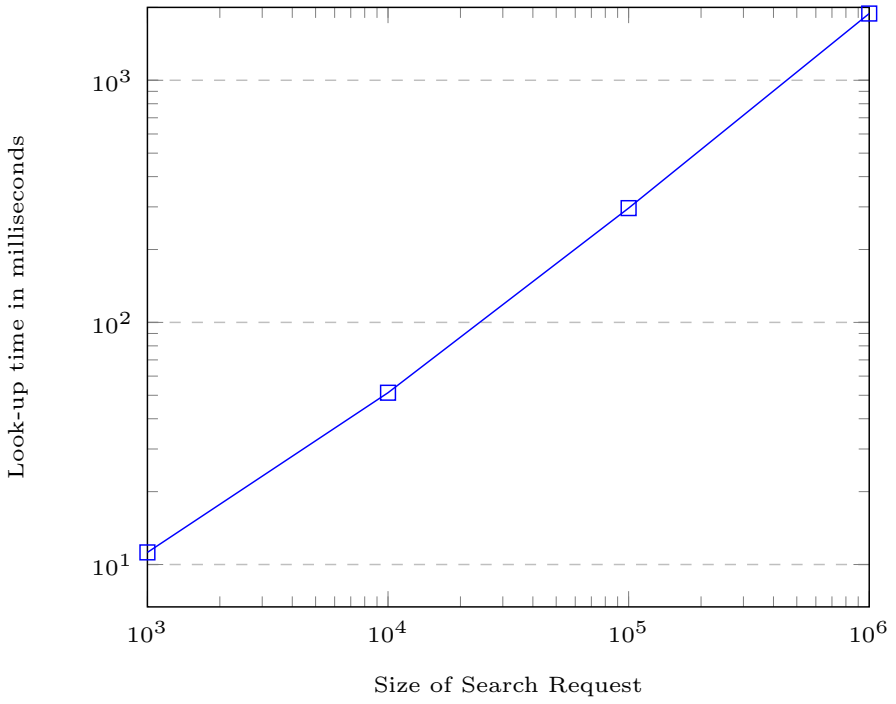


Figure 7.19.: LMDB lookup times for different Search Requests

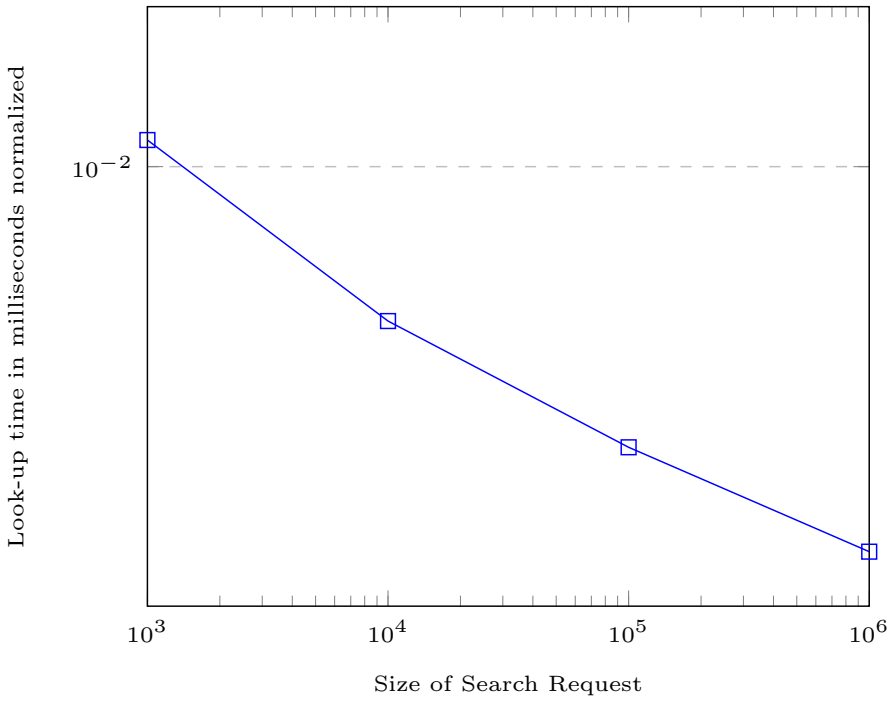


Figure 7.20.: LMDB lookup times for different Search Requests normalized.

specific buckets with for example CRC16. Each bucket is represented with a single database file, holding the values. With this solution, we could build a small cluster of LMDB files. This would allow us to improve insertion times and most likely reduce lookup-times using concurrency.

7.3.3. Comparison of Databases

In this section we want to compare our two evaluated database engines. We evaluated previously Redis in Section 7.3.1 and LMDB in Section 7.3.2. Both databases generally work in our scenario to reverse hash digests of phone numbers – as shown in the two Sections 7.3.1 and 7.3.2.

Here, we compare performance of both databases. While they both reside in RAM, they are built upon vastly different underlying structures. For instance, LMDB is based on B+-Trees (cf. Appendix B) and Redis uses hash tables to store its data. Nonetheless residing in RAM makes both databases viable choices, because RAM is currently the fastest storage excluding registers and cache. In our tests, however, we find different performance for both database engines.

We use our three features to compare both database engines. First, we compare their size growth per inserted records. Then, we compare the lookup times of both tools. Last, we contrast their initiation times for different amount of records.

Size growth. LMDB in this instance handles the included data effectively. It stores the more values in less RAM space. This results in a smaller database, especially in instances with a lower amount of keys inserted. We confirm that LMDB grows linear with more pairs inserted. If we extrapolate our findings, LMDB would need around 9 TB of RAM-space to store 118 billion entries of hash digest and phone number pairs.

In contrast to LMDB, Redis has a higher initial overhead due to the cluster structure. Here the multiple instances of Redis also multiply its overhead. Yet, the size as shown in our

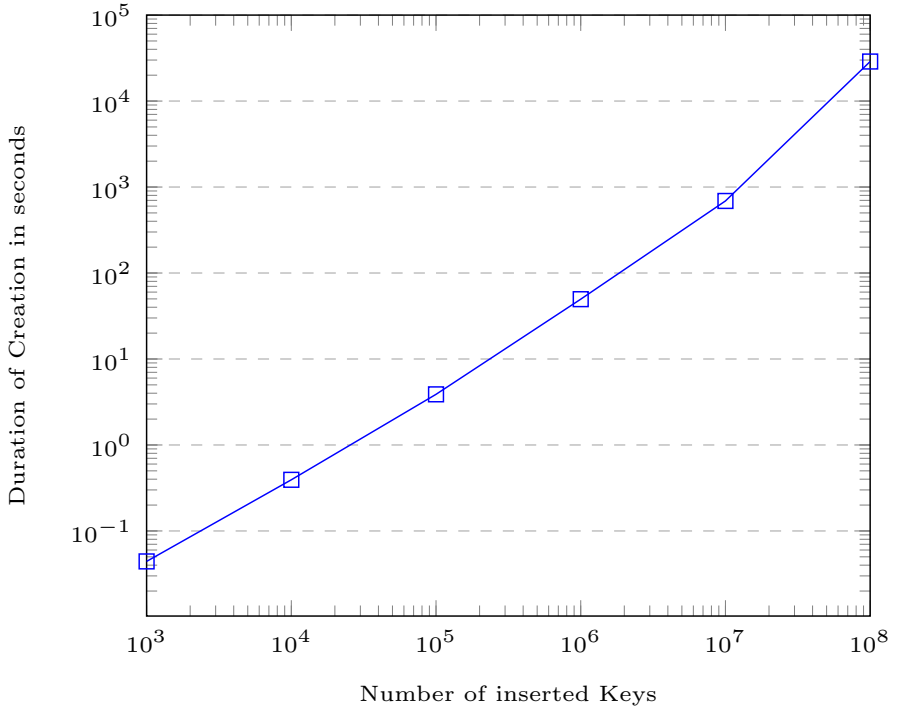


Figure 7.21.: LMDB Initiation time for different amount of keys.

tests also grows linear with expanding number of inserted pairs. By extrapolating those numbers, we would need 19 TB to store the entire number space of 118 billion entries.

We compare their numbers directly in Figure 7.22. The x-axis shows the amount of pairs inserted in the respective database on logarithmic scale. The y-axis shows the normalized size of the database also on a logarithmic scale – meaning the space in kilobyte is used per entry of the database. First thing we see is, that Redis has roughly a 400x factor improvement in comparison to LMDB with 10k inserted records. The factor diminishes once the size of the database grows. In our last measurement with 100M entries, Redis has around double the size of LMDB.

We can see, that LMDB is more space efficient with regards to memory requirements. This is mostly due to 120 nodes deployed in our Redis cluster. We see, that LMDB's initial size overhead doesn't factor much into the size of database. However, Redis' initial size overhead isn't a considerable factor only for a lager number of pairs. Nonetheless, doubled size is still a substantial factor, especially in the context of billions of entries. This would also mean double the cost for a complete deployment of Redis compared to LMDB holding 118 billion entries.

Lookup times. LMDB shows better than constant lookup times (cf. Section 7.3.2.1). In fact, we see for larger search batches even better look-up times. With LMDB we are able to achieve lookup-times below 0.01 ms per key, even getting results of 0.002 ms per key.

With our two tests on Redis (cf. Section 7.3.1.1), we are also able to get constant lookup times. In the first test (max RAM capacity occupied) we achieve around 0.1 ms per searched key. With the second test we are able to get better results. Here, the test shows around 0.03 ms per key.

We directly compare the look-up times for both database engines in Figure 7.23. The graph shows the number of searched keys on the logarithmic x-axis and the normalized lookup-time on the logarithmic y-axis. We also plotted the differences of both databases

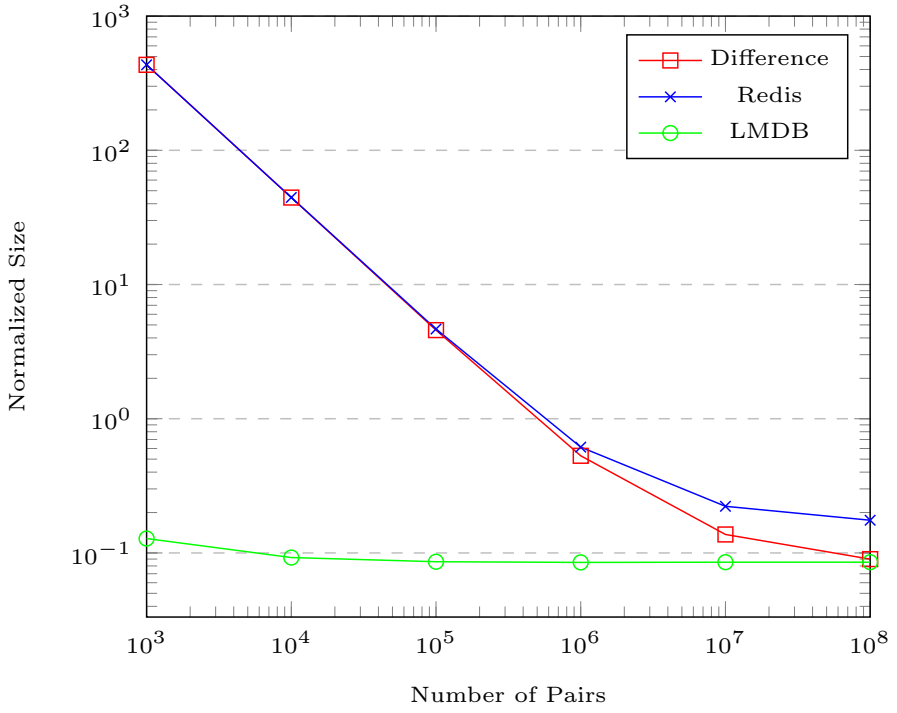


Figure 7.22.: Comparison normalized Size Growth

per number of searched keys. We clearly see, that LMDB is faster than Redis. Redis's and LMDB's constant time for finding values in RAM is true even for a very large amount of pairs inserted. We also see, that LMDB is roughly ten times faster than the results of Redis in [2].

In the Figure 7.23 we also included the difference between the look-up times of Redis and LMDB. We observe here an increasing difference between both database engines. In fact, LMDB has around 20 times faster lookup-times per key for searching 1M keys. This can be attributed to the underlying data structure and minimalistic approach of LMDB.

Initiation time. However, Redis is faster than LMDB in regard of initial creation for larger database sizes. We include in Figure 7.24 the difference in creation time of Redis and LMDB. We can see, that Redis' creation time – including calculation and generating the number/digest pairs – is linear. In contrast to that, LMDB seems to be exponential.

While this is an important observation, a faster creation time is of minor importance, since it is an initial one time cost and not running costs – like look-up times or disk/RAM space. Nonetheless, this is a factor that needs to be further examined, as there could be some additional potential for improvement.

Summary. In conclusion, LMDB shows in our evaluation to have a clear advantage compared to Redis. With half the size and vastly faster lookup-times, LMDB seems to be the better choice for the task of reversing phone number hashes.

However, LMDB shows less efficient behavior than Redis at the database creation time. Furthermore, in our tests we use the cluster feature of Redis. So, we have multiple Redis' instances handling our writes. In contrast, LMDB allows only one writer for one database, which is obviously the bottleneck in addition to the underlying data structure.

For future work, it would be interesting to look into parallelizing LMDB. It would be interesting to see, if the same concept of Redis' CRC16-buckets would make LMDB generally faster – especially for writing the data – and would reduce the initial setup time.

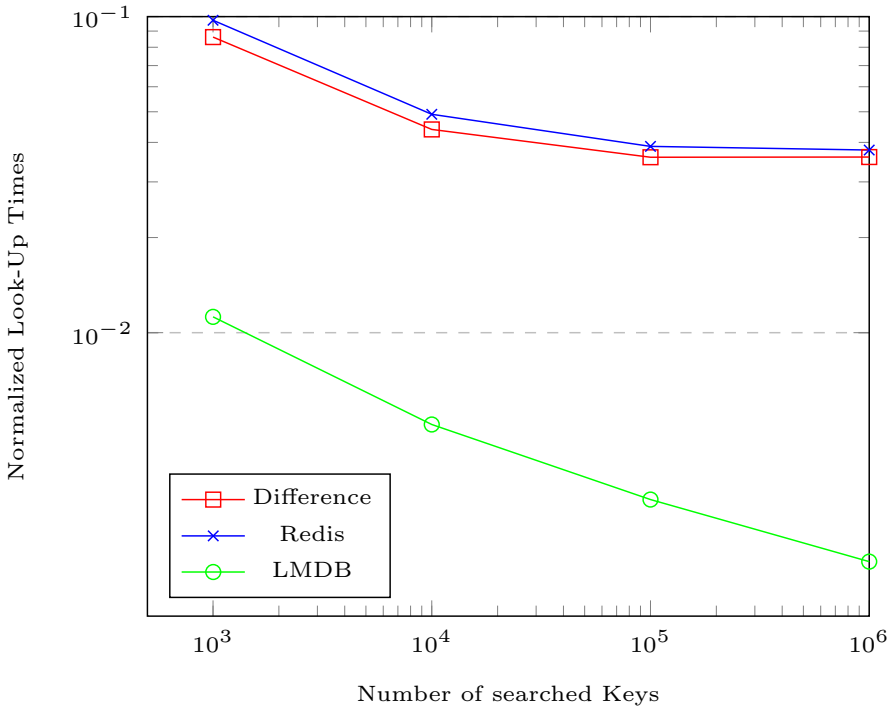


Figure 7.23.: Comparison normalized look-up times

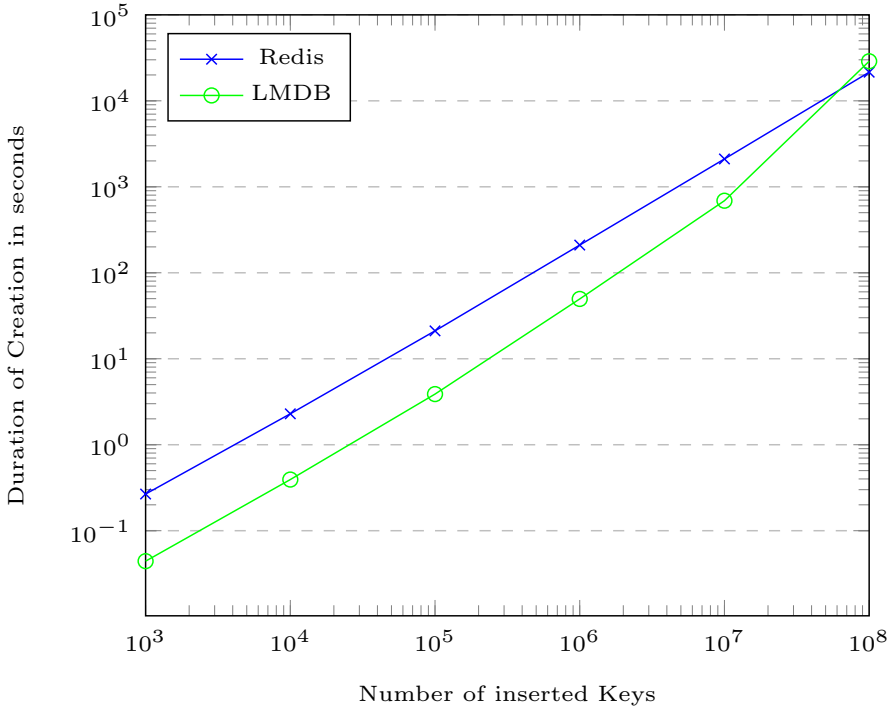


Figure 7.24.: Comparison of Initiation time

7.4. Evaluation of Rainbow Crack

In this section we evaluate hash reversal using rainbow tables. We use our Rainbow Crack test setup called *Manjaro VM* from Section 7.1.5. We explain Rainbow Crack in more detail in Section 2.6. We focus on three aspects in our analysis. First, in Section 7.4.1 we evaluate the hit-rate with different chain lengths and counts. Second, in Section 7.4.2 we reevaluate the analysis of all possible mobile phone numbers. Third, we look into the relation between original number space and needed number of hash calculations to achieve a specific hit-rate in Section 7.4.3.

We focus here on different performance aspects of rainbow tables. As we discuss in Section 2.5, rainbow tables are a trade-off between lookup-times and size of the table. In particular, rainbow tables with longer chains should impose longer lookup times – because one need to retrace the entire chain. With shorter chains, however, one would need to have a higher chain count, which, in turn, would result in a larger table. In this section, we evaluate exactly the impact of these aspects on performance of rainbow tables.

7.4.1. Evaluation Chain Length and Count

We evaluate here the impact of the ratio between chain length and chain count. We assume, that a sum of chain length and chain count is constant and equals ten billion. We then split these 10 billion between length and count in different places to evaluate different size/lookup time performance trade-offs.

First, we create four different tables with the following divisions:

1. 100x100000000
2. 1000x10000000
3. 10000x1000000
4. 100000x100000

We use the corresponding number for the version in our figures. Our first version generates 100000000 chains with the length of 100 reductions. The next three entries just shift the factor ten from number of chains to length of the chains.

The number space of all versions are German mobile phone numbers. We generate all four versions using the command in Listing 2.1.

We can see in Figure 7.25 the generation time of each version. All of them have a similar execution time with minor variation. Nonetheless, we can't observe any meaningful impact on the generation time by varying chain length and count.

After the generation of the rainbow tables, we need to sort them with *rtsort*. We can see on the logarithmic scale in Figure 7.26, that the sorting times of different versions behave according the their respective file-size. In particular, if we save more chains per file, we therefore need longer to sort them.

Now, we generate 10k random hash digests based on our input space. Those hashes are to be reversed with the tool *rcrack*. We visualized our results in Figure 7.27. The success-rate of all versions was above 99%. We see a growth of the lookup-time by a factor of 100. This means, by reducing the size of a table by the factor ten, the lookup times will grow by a factor of 100.

However, we are not able to run the test for version four. If our current data points are a correct indication, the lookup-time for the fourth version would be increased by factor 100

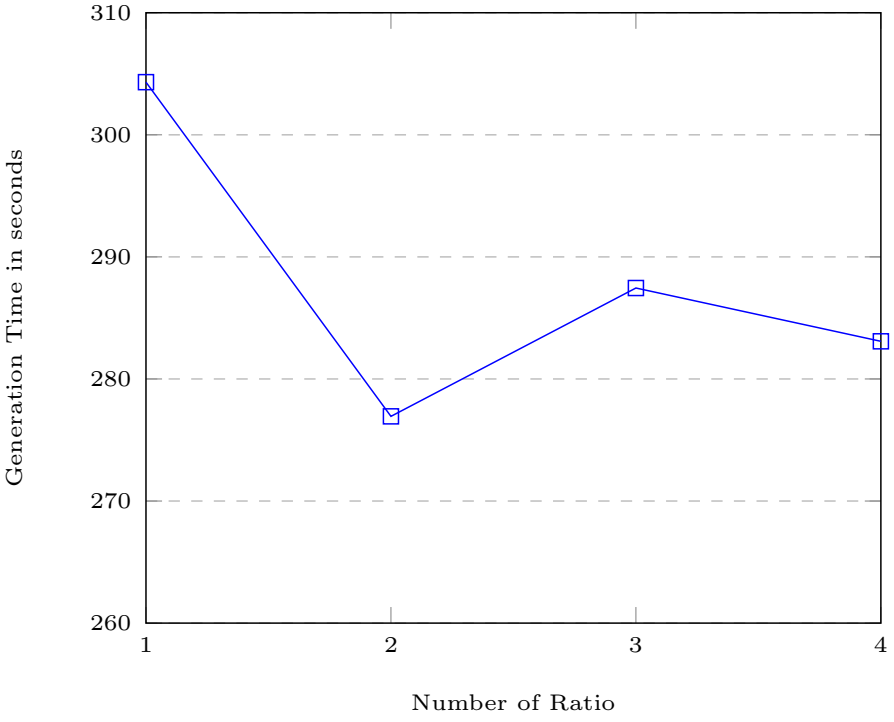


Figure 7.25.: Generation time of different chain lengths and counts.

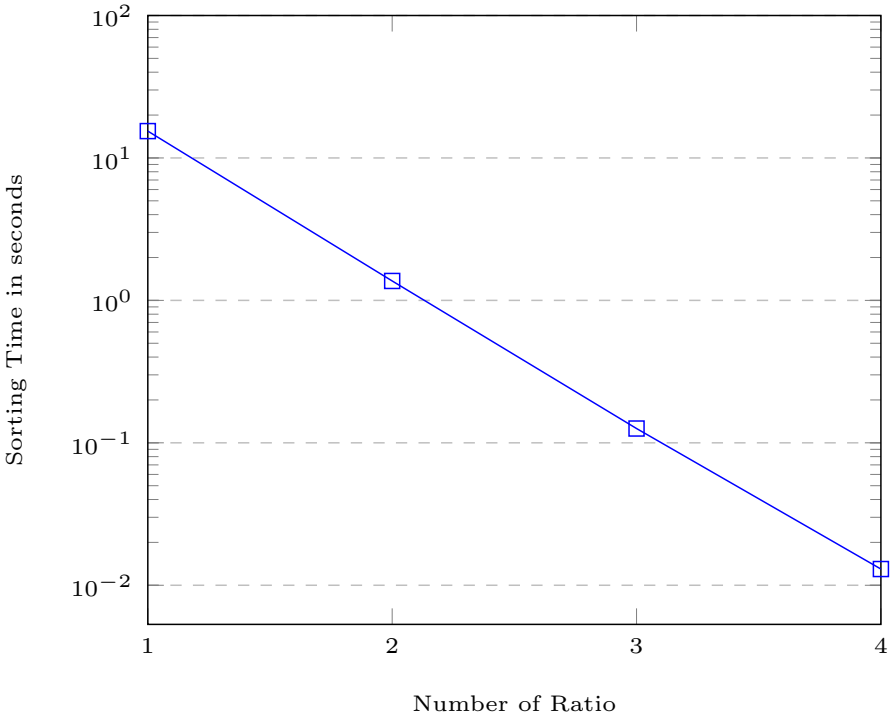


Figure 7.26.: Sorting time of different chain lengths and counts.

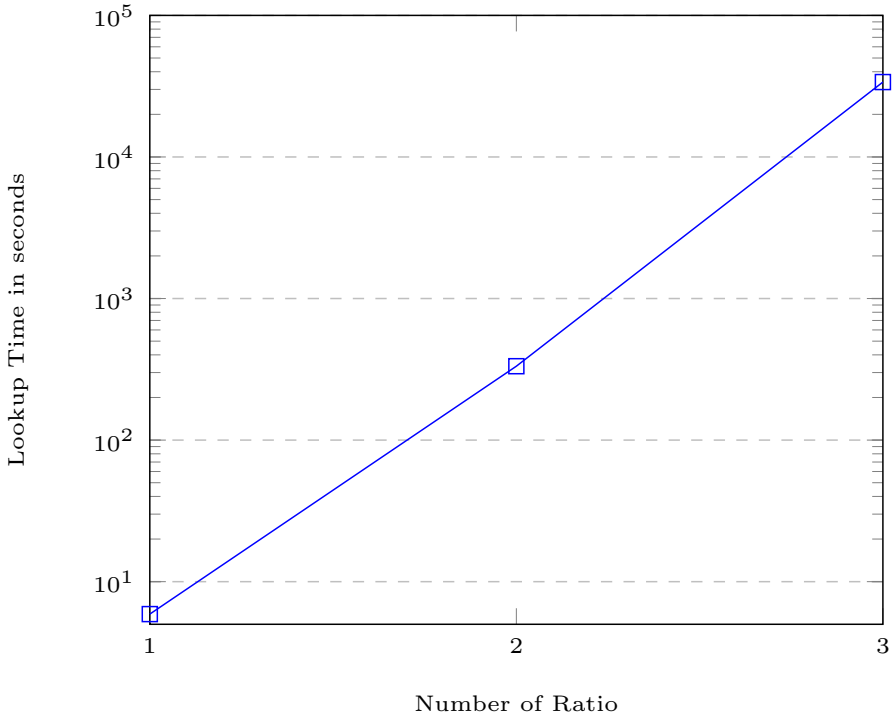


Figure 7.27.: Lookup Time of different chain lengths and counts.

in contrast to version three. So, the lookup would take over 37 days and is not feasible for our purpose of practical hash reversal.

Finally we compare the size impact of the different versions. We plotted our results in Figure 7.28. Here, we can see a predictable behavior of RainbowCrack-NG. As discussed before, each chain is saved with 128 bits (cf. Section 2.6). So, for instance, for our first version, we just multiply the number of chains with their size: $128 \cdot 100000000 = 1.6 \cdot 10^9$ – which is identical to our findings of 1.5 GB for the first version.

7.4.2. Evaluation Reversing All Mobile Phone Number

In this section we evaluate performance of rainbow table for reversing hashes of all mobile phone numbers. We use our test *Manjaro VM* setup from Section 7.1.5 to generate 20 rainbow tables with a chain length of 1k and 100M chains per table. All of those tables target digests from all mobile phone numbers. We need to generate multiple tables to achieve a higher success rate with this approach.

To generate different rainbow tables, we just increment the index number using *rtgen* presented in Section 2.6. The file sizes are constant with 1.5 GB for each index. The index can be seen as a *seed* for each rainbow table, so we don't generate the exact same table.

We first show the characteristics of our tables and then analyze their behavior by reversing 10k random hash digests. However, some aspects of this test are not new, as they are similar to our previous evaluations. Here, we want to show, that an attacker can easily and feasibly reverse hash digests based on world-wide mobile phone numbers.

In Figure 7.29 we visualize the generation time for each rainbow table and the lookup-time of 10k random digests. We can see, that those times vary only marginally between different indexes.

We further examine the hit-rate for 10k random digests in every single Rainbow Table. The Figure 7.30 depicts the success-rate for each index of reversing the same hash digests.

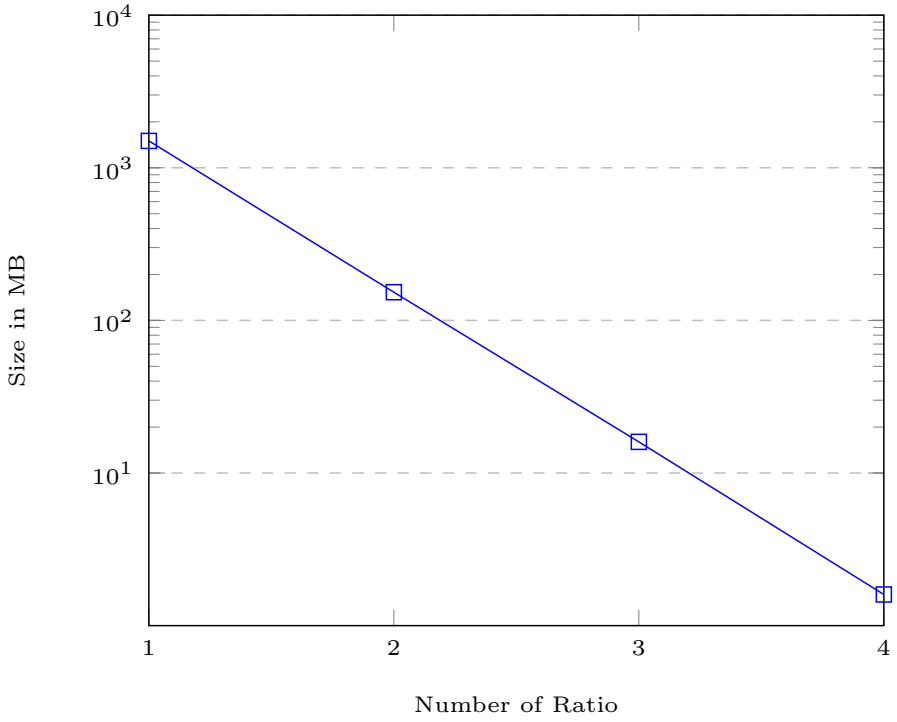


Figure 7.28.: Size of different chain lengths and counts.

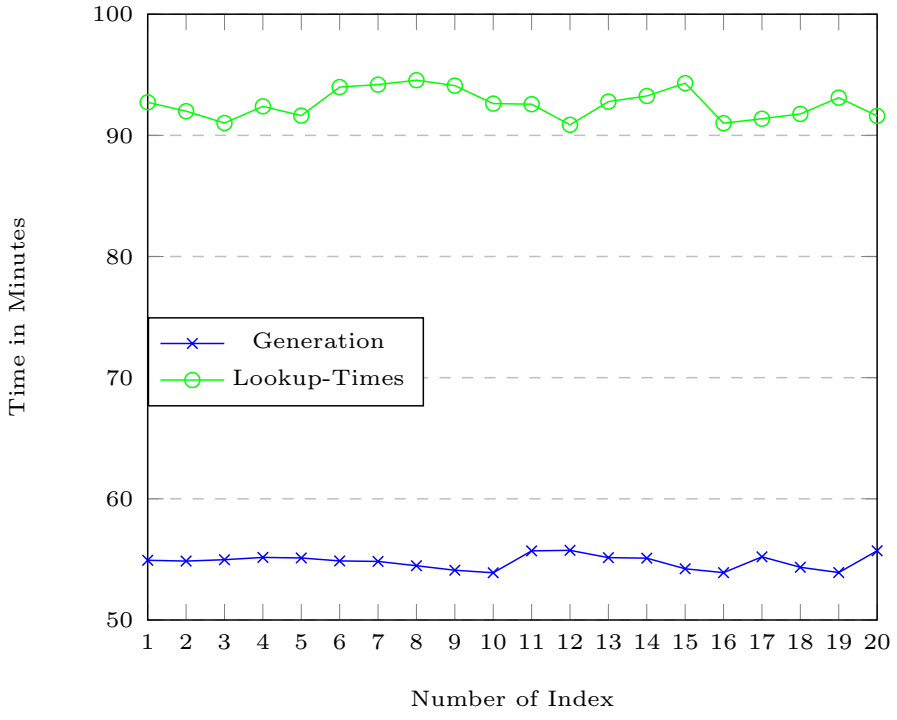


Figure 7.29.: Size of different chain lengths and counts.

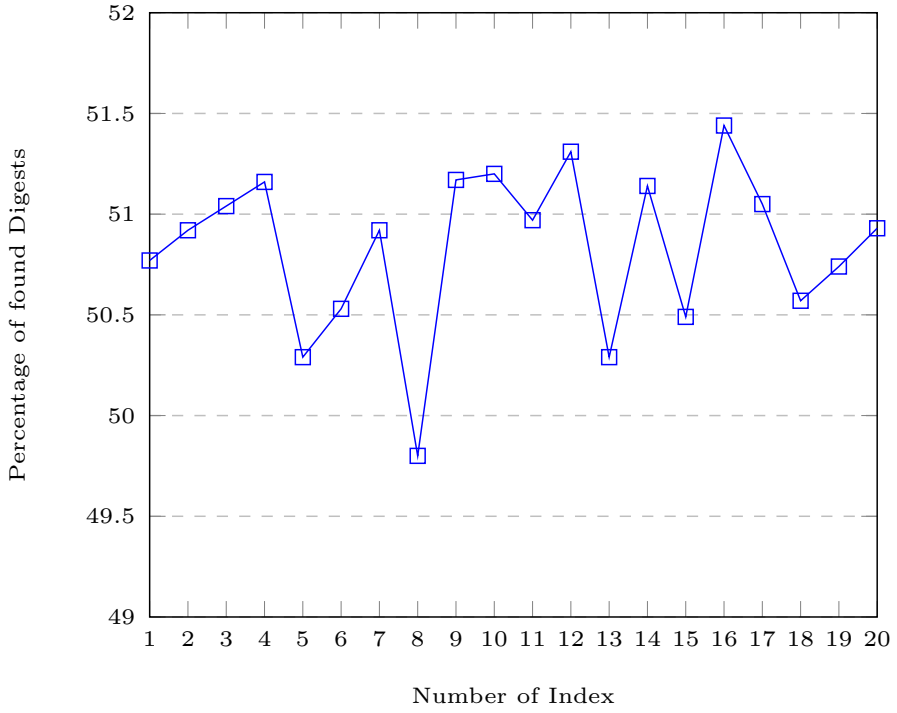


Figure 7.30.: Percentage of found Numbers.

We can observe, that each table itself can reverse around 50% of the searched hashes. We can also observe, that the index has no major effect on the success rate.

We know already, that each generated table has roughly the same success-rate. Now, we also want to look into the actual differences between the tables. For that, we generate 10k random hash digests and reverse the digests with each table independently. We then compare two tables to each other using their solved digests. In other words, we count every successfully reversed digest in both tables. The results are visualized in Table 7.3. We can see in the table, that on average 2585 solved digests are identical within two different rainbow tables. While this indicates a collisions of chains, the collisions are equally distributed. So, while the reduction function creates collisions, there seems no clustering is prevalent.

Equ	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	5,077																			
2	2,572	5,092																		
3	2,546	2,598	5,104																	
4	2,628	2,596	2,614	5,116																
5	2,539	2,562	2,527	2,563	5,029															
6	2,621	2,556	2,557	2,575	2,501	5,053														
7	2,572	2,591	2,576	2,589	2,559	2,599	5,092													
8	2,551	2,521	2,549	2,583	2,495	2,508	2,552	4,980												
9	2,571	2,645	2,598	2,601	2,591	2,555	2,603	2,546	5,117											
10	2,610	2,619	2,658	2,636	2,571	2,591	2,618	2,563	2,613	5,120										
11	2,604	2,550	2,605	2,631	2,535	2,561	2,580	2,521	2,628	2,577	5,097									
12	2,581	2,581	2,632	2,622	2,609	2,591	2,672	2,564	2,638	2,620	2,671	5,131								
13	2,567	2,549	2,573	2,611	2,580	2,561	2,560	2,542	2,558	2,562	2,534	2,566	5,029							
14	2,568	2,630	2,615	2,653	2,557	2,570	2,575	2,555	2,584	2,663	2,650	2,627	2,626	5,114						
15	2,570	2,566	2,575	2,601	2,573	2,569	2,566	2,569	2,568	2,623	2,619	2,603	2,556	2,549	5,049					
16	2,636	2,606	2,573	2,618	2,594	2,591	2,631	2,562	2,583	2,650	2,590	2,598	2,611	2,648	2,581	5,144				
17	2,629	2,646	2,655	2,596	2,544	2,572	2,616	2,548	2,626	2,613	2,564	2,606	2,597	2,588	2,572	2,605	5,105			
18	2,537	2,587	2,560	2,547	2,540	2,582	2,551	2,507	2,580	2,593	2,551	2,666	2,548	2,605	2,572	2,626	2,568	5,057		
19	2,555	2,577	2,630	2,613	2,588	2,554	2,580	2,550	2,623	2,599	2,622	2,593	2,523	2,578	2,522	2,610	2,572	2,537	5,074	
20	2,552	2,578	2,572	2,612	2,512	2,565	2,605	2,569	2,628	2,616	2,606	2,597	2,553	2,582	2,537	2,612	2,600	2,568	2,603	5,093

Table 7.3.: Heatmap of same hashes in different Tables

Another point we can take from Table 7.3 is, that the order of rainbow tables used with *rcrack* doesn't affect the collective success-rate. We use this observation to include all

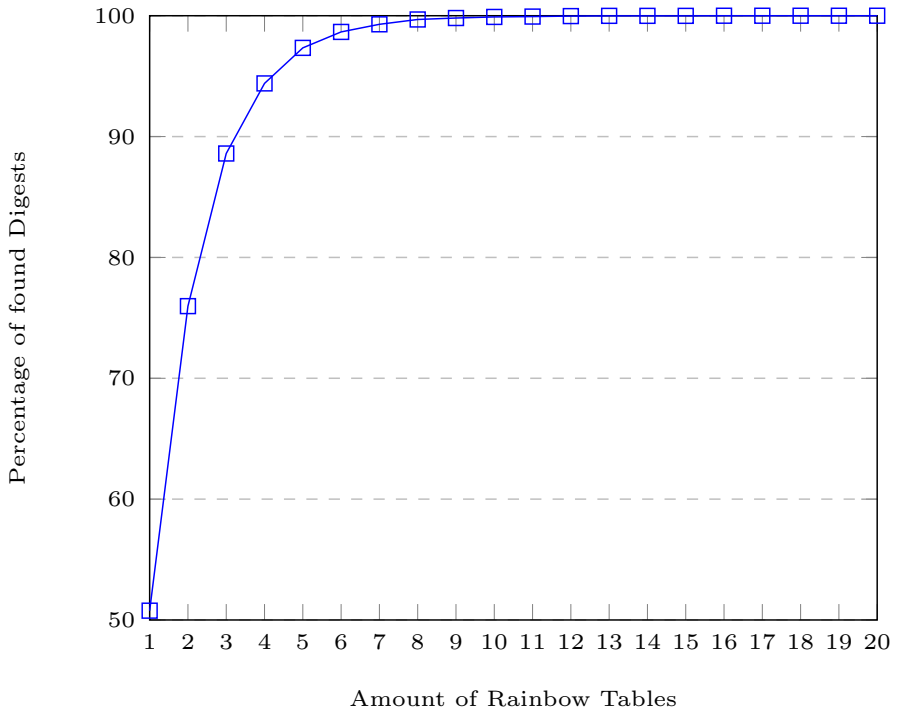


Figure 7.31.: Percentage of found Digests using different amount of Rainbow Tables.

generated rainbow tables to reverse hashes. The different tables are iteratively added to *rcrack*. We want to show with this process the success-rate for different amounts of rainbow tables used in a single lookup.

We first use only one table, then added another one, until we use all 20 generated rainbow tables. We depict our findings in Figure 7.31. Here, we plotted the success-rate using a specific amount of tables. The Figure 7.32 gives an overview of the different look-up times. We can see, that the times only marginally increase and then stagnate with more used tables. We attribute this observation to *rcrack*, where only unsolved digests are searched in the subsequent table.

7.4.3. Evaluation of Size/Performance Trade-off

In this section we look into the relation between number of calculations and the size of the underlying phone number space. In other words, we want to find an estimation of number of calculations needed in order to achieve a certain amount of certainty.

For that, we generate rainbow tables for four different phone number spaces. We use *TF_all* (all France overseas phone numbers), *GN_mobile* (all Guinea mobile phone numbers), *DE_mobile* (all German mobile phone numbers) and *US_all* (all USA phone numbers). We use for all four spaces the chain length of 1000, so we have comparable lookup-times. Nonetheless, the amount of chains varies substantially.

We put together the Table 7.4. Here we can see the assigned number of the space, the prefixes used to generate the space and the size of the resulting number space. We also assume in order to achieve over 99.9% accuracy, we need to generate ten times the number of calculations based on previous tests.

We look first into *TF_all*. Here, we just generate a table with a chain length of 1000 and 100 chains. This is already ten times the targeted number space. Further, we obviously cannot generate randomly distinct 10k hash digests to search. Thus, we only search for

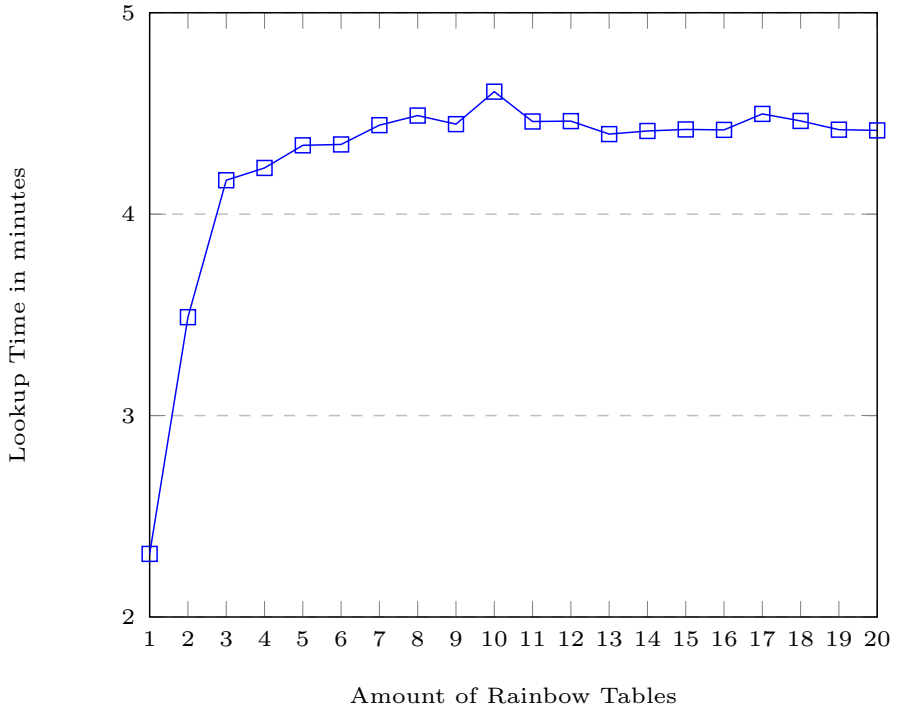


Figure 7.32.: Lookup Times of different amounts of Rainbow Tables.

Table 7.4.: Size of Space

Number	Number Prefix	Cardinality of space
1	*TF_all*	10000
2	*GN_mobile*	11000000
3	*DE_mobile*	538333000
4	*US_all*	5000000000

fewer digests. Our lookup for 6323 different digests resulted in 6206 numbers found – which is a 98.15% success rate. This result is not our initial goal of a success-rate of 99.9%, however the used search space is fundamentally too small.

Next, we use the space of *GN_mobile* for our next test. We create eleven tables – again with a chain length of 1000, but with 10k chains per table. This results in 110 million computations to generate all tables, which is ten times the initial number space. We can see in Figure 7.33, that we achieve over 99.90% success-rate with just nine tables. Thus, we can validate our initial assumption with this experiment.

We further look into the phone number spaces of *DE_mobile* and *US_all*. Both have a chain length of 1000 and have 5 tables generated. For *DE_mobile* we include one million chains per table and for *US_all* 10 million chains. We see in Figure 7.34, that we achieve 99.9% by generating tables with ten times the calculations in comparison to the initial space.

7.5. Discussion

In this section we want to discuss and compare our evaluations of the different hash reversal approaches. Although we tested the different approaches on hardware designed for best performance, we are able to distinguish some key differences of the presented techniques.

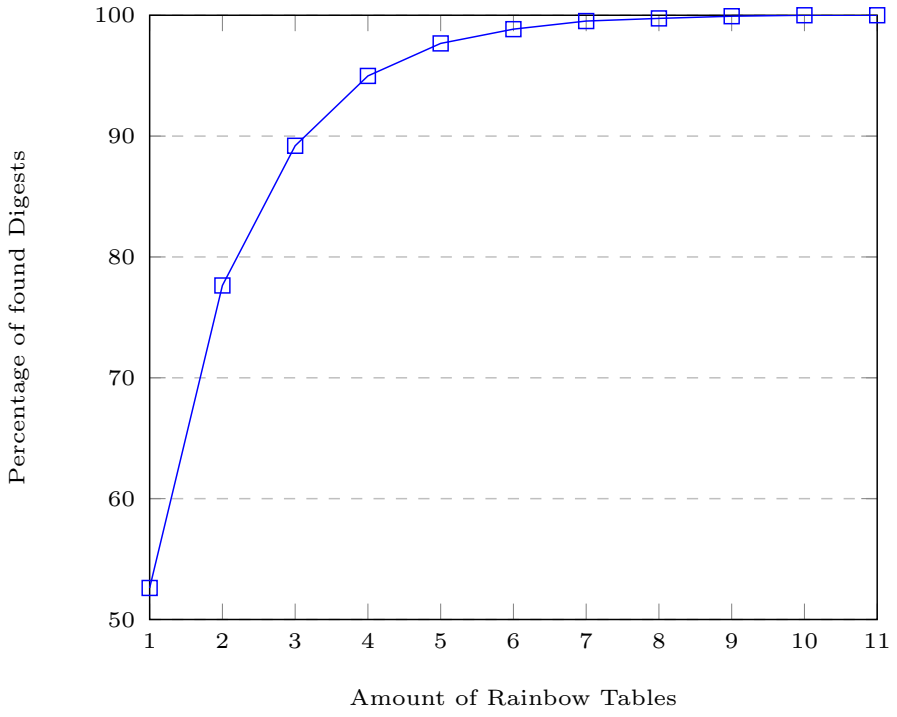


Figure 7.33.: Percentage of found Digests with *GN_mobile*.

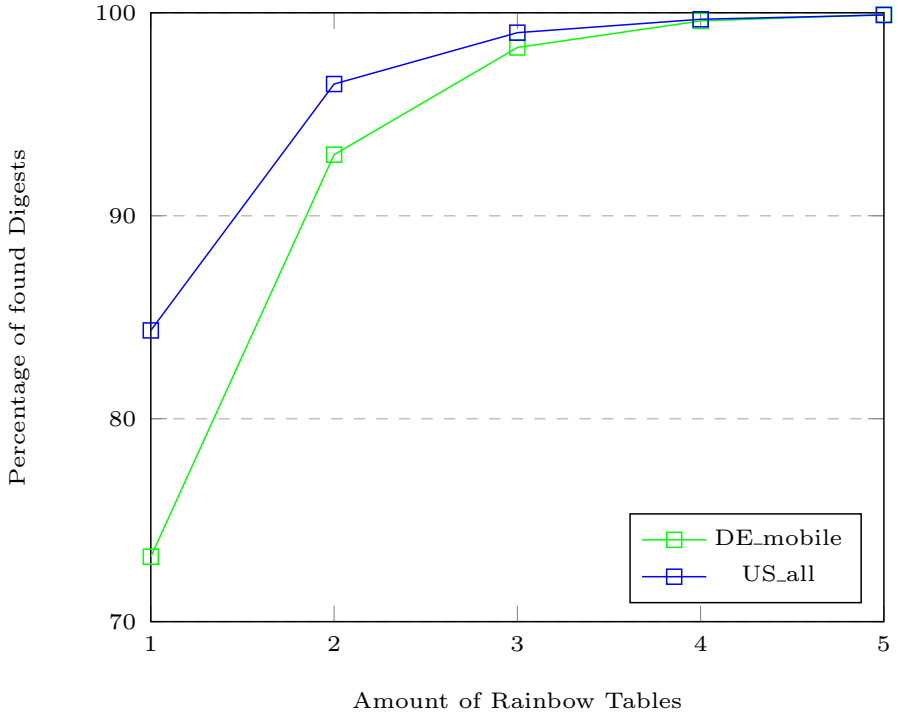


Figure 7.34.: Percentage of found Digests with *DE_mobile* and *US_all*.

Our evaluation of hash databases shows clearly the feasibility of such an attack. While we achieved a better performance with LMDB, we still need around 10 TB of RAM to store all 118 billion mobile phone numbers and their digests. Obviously, this is currently only feasible for a states-actor or a service provider. However, the price tag of 30$ per hour for 12 TB of RAM with a commitment of 3 years makes this approach not economical in comparison to the other approaches.

The use of rainbow tables offers also a viable approach to reverse hashes. We were able to reverse all possible mobile phone numbers with 20 tables achieving a success rate of 100%. While the engineering time of the reduction function and creation time of all tables are only once needed, we still have running costs of a mean lookup time of around 4.5 minutes. Further, this technique shows the feasibility of reversing hashes for an attacker using consumer hardware.

In our opinion, the easiest approach to reverse hashes is brute-force. We were able to reverse all mobile phone numbers in under 100 seconds for different batch sizes using JTR. The tests were done using consumer-grade hardware. With the fast lookup times for larger batch-sizes and a straight-forward engineering approach, our brute-force approach to reverse hash digests shows clearly, that hashing phone numbers only provides a false sense of privacy. Both states-actors/providers and single attackers can reverse the hashes of mobile phone numbers with ease.

8. Conclusion and Future Work

In this thesis, we investigated the privacy problems of contact discovery methods commonly used in mobile messenger apps. We first analyzed and compared contact discovery methods of multiple mobile messenger apps. We discovered, that half of the apps don't provide any anonymization for its users. The other half of messenger apps hash the phone numbers with a simple SHA-algorithm.

We further explored two different hash reversal techniques: Brute-forcing and lookup databases. We proposed for each techniques a generic hash reversal architecture – specific for phone number hashes. We successfully instantiated our brute-force approach with the tools Hashcat and JTR. For the lookup database we also instantiated the generic architecture with two in-memory key-value databases – Redis and LMDB.

In the evaluation we measured three hash reversal techniques. First, we used our brute-force approach to reverse hash digest. We achieved hash reversal times of 93.2 seconds for a million hash digests – accumulating a lookup time of under 0.0932 ms per hash digests. Furthermore, we were able to get near instant lookup times with lookup databases. However, we were limited by our test setups and couldn't include the entire mobile phone number space due to limited RAM available in our test-bed. Last, we evaluated rainbow tables with a specifically crafted reduction function. With this approach we are able to reverse hashes within 4.4 minutes.

In summary of the evaluation, the brute-force approach clearly stands out. The performance of brute-force is twice as fast as rainbow tables, without the overhead of table generation. Further, brute-force doesn't need to be set up like databases – holding so many records in a database is not easily done. In fact, brute-forcing is a straight forward approach, which can be easily adapted for other hashing algorithms or limited to target specific phone number spaces.

In conclusion, we showed, that an attacker can practically reverse hashes based on mobile phone numbers. As a result, the use of hash algorithms for anonymization in mobile messengers only provides marginal privacy improvement.

For future work, we can further improve the rainbow tables. In the current implementation, this technique only uses the CPU to calculate the hashes. We think, that this part can achieve lower generation times and lower lookup times, if a GPU is used instead of a CPU.

The size of the databases can also be further reduced – especially in regards of representation of phone numbers in the database. We currently save the phone numbers as strings.

If we implement a transformation of phone number strings to a phone number integers, we could reduce the needed size substantially. Another factor to consider is, that the numbers are saved in memory. So, we save them in chunks of 4k memory pages. Here, we can further tune the settings of the databases to minimize the empty spaces in allocated pages.

Another idea is to build a cluster with LMDB. There, one can achieve instant lookup times, without experiencing the limitation of a single LMDB-file.

The brute-force approach already operates close to maximum hash-rate of our GPU. However, this approach could theoretically reverse the entire world-wide phone number space in around 13 hours. In future works, we want to setup JTR with this input space.

All three hash reversal techniques should be adapted to other hashing algorithms, such as SHA3 or bcrypt [138]. While the fundamental problem of a limited input space remains, it is not clear what performance impact other hashing functions with larger digests or lower performance have on the three hash reversal techniques.

Bibliography

[1] WhatsApp, "WhatsApp website," 2020 (accessed May 13, 2020). Available: https://web.whatsapp.com/.

[2] C. Hagen, C. Weinert, C. Sendner, A. Dmitrienko, and T. Schneider, "All the numbers are us: Large-scale abuse of contact discovery in mobile messengers." under submission.

[3] T. E. PARLIAMENT, "General data protection regulation," 2016 (accessed May 13, 2020). Available: https://eur-lex.europa.eu/legal-content/EN/TXT/PDF/?uri=CELEX:32016R0679.

[4] Signal, "Signal website," 2020 (accessed May 13, 2020). Available: https://signal.org/en/.

[5] G. E. Moore, "Cramming more components onto integrated circuits," 1965 (accessed May 13, 2020). Available: https://newsroom.intel.com/wp-content/uploads/sites/11/2018/05/moores-law-electronics.pdf.

[6] D. Kales, C. Rechberger, T. Schneider, M. Senker, and C. Weinert, "Mobile private contact discovery at scale." Cryptology ePrint Archive, Report 2019/517, 2019. https://eprint.iacr.org/2019/517.

[7] Ágnes Kiss, J. Liu, T. Schneider, N. Asokan, and B. Pinkas, "Private set intersection for unequal set sizes with mobile applications," *Proceedings on Privacy Enhancing Technologies*, vol. 2017, no. 4, pp. 177 – 197, 2017.

[8] Telegram, "Telegram website," 2020 (accessed May 13, 2020). Available: https://telegram.org/.

[9] Snapchat, "Snapchat website," 2020 (accessed May 13, 2020). Available: https://www.snapchat.com/.

[10] Threema, "Threema official website," 2020 (accessed February 18, 2020). Available: https://threema.ch/en.

[11] WeChat, "WeChat website," 2020 (accessed May 13, 2020). Available: https://wx.qq.com/?lang=en_US.

[12] Viber, "Viber," 2020 (accessed May 13, 2020). Available: https://www.viber.com/en/.

[13] W. S. GmbH, "Wire website," 2020 (accessed February 18, 2020). Available: https://wire.com/.

[14] J. 'atom' Steube and G. 'matrix' Gristina, "Hashcat website," 2019 (accessed December 08, 2019). Available: https://hashcat.net/hashcat/.

[15] S. Designer, "John the Ripper website," 2019 (accessed December 08, 2019). Available: https://www.openwall.com/john/.

[16] Redis, "Redis official website," 2020 (accessed February 18, 2020). Available: https://redis.io/.

[17] H. Chu, "LMDB website," 2015 (accessed May 13, 2020). Available: http://www.lmdb.tech/doc/.

[18] S. Schindler, "Creating rainbow tables for mobile phone numbers," 2019 (accessed May 13, 2020). Master practicum at University of Würzburg.

[19] P. Oechslin, "Making a faster cryptanalytic time-memory trade-off," in *Advances in Cryptology - CRYPTO 2003* (D. Boneh, ed.), (Berlin, Heidelberg), pp. 617–630, Springer Berlin Heidelberg, 2003.

[20] ITU-T, "The international public telecommunication numbering plan," 2010 (accessed December 08, 2019). Available: https://www.itu.int/rec/dologin_pub.asp?lang=e&id=T-REC-E.164-201011-I!!PDF-E&type=items.

[21] Bundesnetzagentur, "Nummerierungskonzept 2011," 2019 (accessed December 08, 2019). Available: https://www.bundesnetzagentur.de/SharedDocs/Downloads/DE/Sachgebiete/Telekommunikation/Unternehmen_Institutionen/Nummerierung/Nummerierungskonzept/Nummerierungskonzept2011pdf.pdf?__blob=publicationFile.

[22] M. Dörrbecker, "Phone number setup," 2017 (accessed December 08, 2019). Available: https://commons.m.wikimedia.org/wiki/File:Phone_number_setup.png.

[23] Google, "Google's common Java, C++ and JavaScript library for parsing, formatting, and validating international phone numbers.," 2017 (accessed December 08, 2019). Available: https://github.com/google/libphonenumber.

[24] Signal, "Signal's use of goolge-libphonenumber," 2020 (accessed February 18, 2020). Available: https://github.com/signalapp/Signal-Desktop/blob/800c7ed31d271fa384cb4f718f678d19e604683a/package.json#L91.

[25] NIST, "Secure Hash Standard," 1995 (accessed February 18, 2020). Available: https://csrc.nist.gov/publications/detail/fips/180/1/archive/1995-04-17.

[26] P. E. Jones and r. Donald E. Eastlake, "RFC 3174 SHA1," 2001 (accessed February 18, 2020). Available: https://tools.ietf.org/html/rfc3174.

[27] H. Tiwari, "Merkle-damgård construction method and alternatives: A review," *Journal of Information and Organizational Sciences*, vol. 41, pp. 283–304, 12 2017.

[28] NIST, "SHA-3 standard: Permutation-based hash and extendable-output functions," 2015 (accessed February 18, 2020). Available: https://csrc.nist.gov/publications/detail/fips/202/final.

[29] A. Joux, "Multicollisions in iterated hash functions. application to cascaded constructions," in *Advances in Cryptology – CRYPTO 2004* (M. Franklin, ed.), (Berlin, Heidelberg), pp. 306–316, Springer Berlin Heidelberg, 2004.

[30] J. Kelsey and B. Schneier, "Second preimages on n-bit hash functions for much less than 2nwork," in *Advances in Cryptology – EUROCRYPT 2005* (R. Cramer, ed.), (Berlin, Heidelberg), pp. 474–490, Springer Berlin Heidelberg, 2005.

[31] J. Kelsey and T. Kohno, "Herding hash functions and the Nostradamus attack," in *Advances in Cryptology - EUROCRYPT 2006* (S. Vaudenay, ed.), (Berlin, Heidelberg), pp. 183–200, Springer Berlin Heidelberg, 2006.

[32] Redis, "Redis GET documentation," 2009 (accessed February 18, 2020). Available: https://redis.io/commands/get.

[33] S. Sanfilippo, "Redis source code referencing memory hash tables using chaining for collision handling," 2009 (accessed February 18, 2020). Available: https://github.com/antirez/redis/blob/13707f988b8095914ca95ac6a836dad1573ea5bd/src/dict.c#L3-L6.

[34] MySQL, "MySQL official website," 2020 (accessed February 18, 2020). Available: https://www.mysql.com/.

[35] J. Stolfi, "Hash table example," 2019 (accessed December 08, 2019). Available: https://commons.wikimedia.org/wiki/File:Hash_table_3_1_1_0_1_0_0_SP.svg.

[36] M. C. Borja and J. Haigh, "The birthday problem," *Significance*, vol. 4, no. 3, pp. 124–127, 2007.

[37] H. Chu, "MDB: A memory-mapped database and backend for OpenLDAP," 2011 (accessed February 18, 2020). Available: https://ldapcon.org/2011/downloads/chu-paper.pdf.

[38] M. Hellman, "A cryptanalytic time-memory trade-off," vol. 26, pp. 401–406, July 1980.

[39] Z. Shuanglei, "RainbowCrack-NG Github," 2020 (accessed May 13, 2020). Available: https://github.com/inAudible-NG/RainbowCrack-NG.

[40] P. D. . Telegram, "400 million users, 20,000 stickers, quizzes 2.0 and 400k EUR for creators of educational tests," 2020 (accessed February 18, 2020). Available: https://telegram.org/blog/400-million.

[41] Telegram, "API for importing contacts to Telegram," 2014 (accessed February 18, 2020). Available: https://core.telegram.org/method/contacts.importContacts.

[42] V. Valtman, "API for importing contacts to Telegram by third-party client," 2019 (accessed February 18, 2020). Available: https://github.com/kenorb-contrib/tgl/blob/57f1bc41ae13297e6c3e23ac465fd45ec6659f50/queries.c#L2556.

[43] J. K. . WhatsApp, "400 million stories," 2013 (accessed February 18, 2020). Available: https://web.archive.org/web/20140412064444/http://blog.whatsapp.com/index.php/2013/12/400-million-stories/?lang=de.

[44] R. Albergotti, D. MacMillan, and E. M. Rusli, "Facebook to pay 19 billion for WhatsApp," 2014 (accessed February 18, 2020). Available: https://www.wsj.com/articles/facebook-to-buy-whatsapp-for-16-billion-1392847766.

[45] W. blog, "Two billion WhatsApp user world-wide," 2020 (accessed February 25, 2020). Available: https://blog.whatsapp.com/two-billion-users-connecting-the-world-privately.

[46] I. A. N. SERVICE, "WhatsApp registers over 5 billion installs on Google Play Store," 2020 (accessed February 25, 2020). Available: https://tech.hindustantimes.com/tech/news/whatsapp-registers-over-5-billion-installs-on-google-play-store-story-jgg9GSYtRWHwZeDIUwK7GL.html.

[47] Github, "Third party desktop WhatsApp clients," 2020 (accessed February 25, 2020). Available: https://github.com/topics/whatsapp-desktop.

[48] R. Zaikin and O. V. . black hat 2019, "Reverse engineering WhatsApp encryption for chat manipulation and more," 2019 (accessed February 25, 2020). Available: https://i.blackhat.com/USA-19/Wednesday/us-19-Zaikin-Reverse-Engineering-WhatsApp-Encryption-For-Chat-Manipulation-And-More.pdf.

[49] WhatsApp, "WhatsApp's privacy notice to clarify, that phone numbers are stored as hashes," 2012 (accessed February 18, 2020). Available: https://www.whatsapp.com/legal?doc=privacy-policy&version=20120707.

[50] K. Hill, "Facebook recommended that this psychiatrist's patients friend each other," 2016 (accessed February 18, 2020). Available: https://splinternews.com/facebook-recommended-that-this-psychiatrists-patients-f-1793861472.

[51] WhatsApp, "Whatsapp Apple App Store US," 2020 (accessed May 13, 2020). Available: https://apps.apple.com/us/app/whatsapp-messenger/id310633997.

[52] Snapchat, "Snap Inc. announces first quarter 2020 financial results," 2020 (accessed May 13, 2020). Available: https://investor.snap.com/news-releases/2020/04-21-2020-210949737.

[53] S. Inc., "Snap KIT: Developer tools for snapchat," 2020 (accessed May 13, 2020). Available: https://kit.snapchat.com/.

[54] Snapchat, "Snapchat's privacy policy," 2019 (accessed February 18, 2020). Available: https://www.snap.com/en-US/privacy/privacy-policy.

[55] Snapchat, "Snapchat support creating an account," 2020 (accessed May 13, 2020). Available: https://support.snapchat.com/en-US/a/account-setup.

[56] Google, "Threema in Google Play Store," 2020 (accessed February 18, 2020). Available: https://play.google.com/store/apps/details?id=ch.threema.app&hl=en.

[57] Threema, "Threema press release," 2020 (accessed May 13, 2020). Available: https://threema.ch/press-files/1_press_info/Press-Info_Threema_EN.pdf.

[58] Threema, "Threema whitepaper," 2019 (accessed February 18, 2020). Available: https://threema.ch/press-files/2_documentation/cryptography_whitepaper.pdf.

[59] H. Gierow, "Sicherheitsforscher öffnen Threema-Blackbox," 2016 (accessed February 18, 2020). Available: https://www.golem.de/news/reverse-engineering-sicherheitsforscher-oeffnen-threema-blackbox-1612-125286.html.

[60] cryptix, "Open-source implementation of the Threema protocol in Go.," 2019 (accessed February 18, 2020). Available: https://github.com/o3ma/o3.

[61] 33C3, "A look into the mobile messaging black box," 2016 (accessed February 18, 2020). Available: https://media.ccc.de/v/33c3-8062-a_look_into_the_mobile_messaging_black_box.

[62] Threema, "Which data gets stored at Threema?," 2020 (accessed February 18, 2020). Available: https://threema.ch/en/faq/data.

[63] N. J. . Technode, "WeChat now has over 1 billion active monthly users worldwide," 2018 (accessed May 13, 2020). Available: https://technode.com/2018/03/05/wechat-1-billion-users/.

[64] Tencent, "Tencent legal statement," 2020 (accessed May 13, 2020). Available: https://www.tencent.com/en-us/statement.html.

[65] S. M. . Techinasia, "Tencent responds in case of apparent WeChat censorship," 2013 (accessed May 13, 2020). Available: https://www.techinasia.com/tencent-responds-wechat-censoring-sensitive-words.

[66] WeChat, "Wechat API," 2020 (accessed May 13, 2020). Available: https://developers.weixin.qq.com/miniprogram/en/dev/api/.

[67] wechat4u, "Github wechat4u.js," 2020 (accessed May 13, 2020). Available: https://github.com/nodeWechat/wechat4u.

[68] WeChat, "WeChat's FAQ: Does WeChat import all contacts on my phone automatically?," 2019 (accessed February 18, 2020). Available: https://www.wechat.co.za/faq/does-wechat-import-all-contacts-on-my-phone-automatically/.

[69] WeChat, "WeChat's privacy policy on how data is processed," 2019 (accessed February 18, 2020). Available: https://www.wechat.com/en/privacy_policy.html#pp_how.

[70] C. S. . TechCrunch, "Japanese internet giant Rakuten acquires Viber for 900M Dollar," 2014 (accessed May 13, 2020). Available: https://techcrunch.com/2014/02/13/japanese-internet-giant-rakuten-acquires-viber-for-900m/.

[71] Rekuten, "Rakuten global website," 2020 (accessed May 13, 2020). Available: https://global.rakuten.com/corp/.

[72] J. J. . Statista, "Ranking of the most used Android apps in Ukraine in June 2019," 2019 (accessed May 13, 2020). Available: https://www.statista.com/statistics/1029262/most-used-android-apps-ukraine/.

[73] J. C. . Statista, "Number of unique Viber user IDs from June 2011 to March 2019," 2019 (accessed May 13, 2020). Available: https://www.statista.com/statistics/316414/viber-messenger-registered-users/.

[74] Viber, "Viber encryption overview," 2018 (accessed February 18, 2020). Available: https://www.viber.com/app/uploads/viber-encryption-overview.pdf.

[75] Viber, "Viber API," 2020 (accessed May 13, 2020). Available: https://developers.viber.com/docs/api/rest-bot-api/.

[76] Viber, "Viber's FAQ: Manage your Viber contacts," 2019 (accessed February 18, 2020). Available: https://help.viber.com/en/article/manage-your-viber-contacts.

[77] H. James, "I refuse to sign up for Viber, here's why.," 2019 (accessed February 18, 2020). Available: https://haydenjames.io/i-refuse-to-sign-up-for-viber-heres-why/.

[78] L. W. . cnet, "Viber desktop app puts phone chat on your computer," 2013 (accessed May 13, 2020). Available: https://www.cnet.com/news/viber-desktop-app-puts-phone-chat-on-your-computer/.

[79] Microsoft, "Skype website," 2020 (accessed February 18, 2020). Available: https://www.skype.com/.

[80] Canadajournal, "'wire': Skype co-founder Janus Friis launches ultra-private messaging, with video," 2016 (accessed February 18, 2020). Available: https://canadajournal.net/world/wire-skype-co-founder-janus-friis-launches-ultra-private-messaging-video-44561-2016/.

[81] W. S. GmbH, "Wire security whitepaper," 2018 (accessed February 18, 2020). Available: https://wire-docs.wire.com/download/Wire+Security+Whitepaper.pdf.

[82] W. S. GmbH, "Wire privacy whitepaper," 2018 (accessed February 18, 2020). Available: https://wire-docs.wire.com/download/Wire+Privacy+Whitepaper.pdf.

[83] Google, "Wire app in Google Play Store," 2020 (accessed February 18, 2020). Available: https://play.google.com/store/apps/details?id=com.wire&hl=en.

[84] A. . Fossmint, "Wire messaging software has gone open-source invites devs to contribute," 2016 (accessed February 18, 2020). Available: https://www.fossmint.com/wire-messaging-software-has-gone-open-source-invites-devs-to-contribute/.

[85] W. . Medium, "Wire server code now 100% open source — the journey continues," 2017 (accessed February 18, 2020). Available: https://medium.com/@wireapp/wire-server-code-now-100-open-source-the-journey-continues-88e24164309c.

[86] O. Foundation, "Electron," 2020 (accessed February 18, 2020). Available: https://www.electronjs.org/.

[87] K. Cohn-Gordon, C. Cremers, B. Dowling, L. Garratt, and D. Stebila, "A Formal Security Analysis of the Signal Messaging Protocol." Cryptology ePrint Archive, Report 2016/1013, 2016. https://eprint.iacr.org/2016/1013.

[88] Signal, "Signal's use of SHA1 to anonymize recipients," 2019 (accessed February 18, 2020). Available: https://github.com/signalapp/Signal-iOS/blob/a1c6b7ec154ca999ed9ef9d4a1d095597b610fb5/SignalServiceKit/src/Contacts/OWSContactDiscoveryOperation.swift#L45.

[89] moxie0, "Signal's blog post to push for Intel SGX Contact Discovery," 2017 (accessed February 18, 2020). Available: https://signal.org/blog/private-contact-discovery/.

[90] T. F. LLC, "Telegram logo," 2019 (accessed December 08, 2019). Available: https://commons.wikimedia.org/wiki/File:Telegram_2019_Logo.svg.

[91] WhatsApp, "WhatsApp logo," 2015 (accessed December 08, 2019). Available: https://commons.wikimedia.org/wiki/File:WhatsApp.svg.

[92] S. Inc., "Snapchat logo," 2019 (accessed December 08, 2019). Available: https://en.wikipedia.org/wiki/File:Snapchat_logo.svg.

[93] Threema, "Threema logo," 2014 (accessed December 08, 2019). Available: https://commons.wikimedia.org/wiki/File:Threema.png.

[94] WeChat, "Wechat logo," 2011 (accessed December 08, 2019). Available: https://commons.wikimedia.org/wiki/File:WeChat_logo_Android.png.

[95] F. Alexis, "Viber logo," 2016 (accessed December 08, 2019). Available: https://commons.wikimedia.org/wiki/File:Antu_viber.svg.

[96] W. S. GmBH, "Wire logo," 2020 (accessed February 18, 2020). Available: https://lh3.googleusercontent.com/gcw6-BzPp0HX3lSFwlsBOZ5ZKKkBXAzPRtco7aAYqe0gLK3jrWy_kCZ9uvN2zQLSoV8.

[97] T. Reinhard, "Signal logo," 2015 (accessed December 08, 2019). Available: https://de.wikipedia.org/wiki/Datei:Signal_Blue_Icon.png.

[98] B. A. Huberman, M. Franklin, and T. Hogg, "Enhancing privacy and trust in electronic communities," in *Proceedings of the 1st ACM Conference on Electronic Commerce*, EC '99, (New York, NY, USA), p. 78–86, Association for Computing Machinery, 1999.

[99] Y. Huang, D. Evans, and J. Katz, "Private set intersection: Are garbled circuits better than custom protocols?," in *19th Annual Network and Distributed System Security Symposium, NDSS 2012, San Diego, California, USA, February 5-8, 2012*, The Internet Society, 2012.

[100] Y. Ishai, J. Kilian, K. Nissim, and E. Petrank, "Extending oblivious transfers efficiently," 06 2003.

[101] C. Dong, L. Chen, and Z. Wen, "When private set intersection meets big data: An efficient and scalable protocol," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security*, CCS '13, (New York, NY, USA), p. 789–800, Association for Computing Machinery, 2013.

[102] C. Hazay and Y. Lindell, "Constructions of truly practical secure protocols using standardsmartcards," in *Proceedings of the 15th ACM Conference on Computer and Communications Security*, CCS '08, (New York, NY, USA), p. 491–500, Association for Computing Machinery, 2008.

[103] B. Pinkas, T. Schneider, and M. Zohner, "Faster private set intersection based on ot extension." Cryptology ePrint Archive, Report 2014/447, 2014. `https://eprint.iacr.org/2014/447`.

[104] B. Pinkas, M. Rosulek, N. Trieu, and A. Yanai, "Spot-light: Lightweight private set intersection from sparse ot extension," in *Advances in Cryptology – CRYPTO 2019* (A. Boldyreva and D. Micciancio, eds.), (Cham), pp. 401–431, Springer International Publishing, 2019.

[105] H. Chen, K. Laine, and P. Rindal, "Fast private set intersection from homomorphic encryption," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, (New York, NY, USA), p. 1243–1255, Association for Computing Machinery, 2017.

[106] H. Chen, Z. Huang, K. Laine, and P. Rindal, "Labeled psi from fully homomorphic encryption with malicious security," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, (New York, NY, USA), p. 1223–1237, Association for Computing Machinery, 2018.

[107] E. Kim, K. Park, H. Kim, and J. Song, "I've got your number: - harvesting users' personal data via contacts sync for the kakaotalk messenger," in *Information Security Applications - 15th International Workshop, WISA 2014, Jeju Island, Korea, August 25-27, 2014. Revised Selected Papers* (K. H. Rhee and J. H. Yi, eds.), vol. 8909 of *Lecture Notes in Computer Science*, pp. 55–67, Springer, 2014.

[108] J. Kim, K. Kim, J. Cho, H. Kim, and S. Schrittwieser, "Hello, facebook! here is the stalkers' paradise!: Design and analysis of enumeration attack using phone numbers on facebook," in *Information Security Practice and Experience* (J. K. Liu and P. Samarati, eds.), (Cham), pp. 663–677, Springer International Publishing, 2017.

[109] S. Schrittwieser, P. Frühwirt, P. Kieseberg, M. Leithner, M. Mulazzani, M. Huber, and E. R. Weippl, "Guess who's texting you? evaluating the security of smartphone messaging applications," in *NDSS*, 2012.

[110] J. M. G. Estrada, "WhatsApp-Scraping," 2020 (accessed February 18, 2020). Available: `https://github.com/JMGama/WhatsApp-Scraping`.

[111] S. PJ, "WhatsApp crawler," 2020 (accessed February 18, 2020). Available: `https://gitlab.com/jishnutp/whatsapp-crawler`.

[112] R. Project, "RainbowCrack website," 2020 (accessed February 18, 2020). Available: http://project-rainbowcrack.com/.

[113] srip / flaticon.com, "CSV icon," 2020 (accessed February 18, 2020). Available: https://www.flaticon.com/free-icon/csv-file-format_1263920.

[114] Y. L. . iconfinder.com, "hawcons icons," 2020 (accessed February 18, 2020). Available: https://www.iconfinder.com/iconsets/hawcons.

[115] Hashtopolis, "Hashtopolis icon," 2016 (accessed May 13, 2020). Available: https://github.com/s3inlc/hashtopolis/blob/master/src/static/logo.png.

[116] free-icons download.net, "Look-Up icon," 2016 (accessed May 13, 2020). Available: http://www.free-icons-download.net/lookup-icon-65553/.

[117] srip, "TXT icon," 2020 (accessed May 13, 2020). Available: https://www.flaticon.com/free-icon/txt_1263942.

[118] H. Wiki, "Generic hash types of Hashcat," 2020 (accessed May 13, 2020). Available: https://hashcat.net/wiki/doku.php?id=example_hashes.

[119] J. Foster, "Database icon," 2020 (accessed May 13, 2020). Available: https://icon-icons.com/icon/office-database/103574.

[120] Dormando, "Memcached," 2018 (accessed May 13, 2020). Available: https://memcached.org/.

[121] B. Inc., "BigBlueButton architecture with Redis," 2020 (accessed May 13, 2020). Available: https://docs.bigbluebutton.org/2.2/architecture.html.

[122] F. . flaticon.com, "Python icon," 2020 (accessed February 18, 2020). Available: https://www.flaticon.com/free-icon/python_1822899?term=python&page=1&position=2.

[123] Redis, "Redis mass insertion," 2020 (accessed May 13, 2020). Available: https://redis.io/topics/mass-insert.

[124] vipshop, "HIREDIS-VIP GitHub page," 2017 (accessed May 13, 2020). Available: https://github.com/vipshop/hiredis-vip.

[125] C. Prioglio, "Redis icon," 2017 (accessed May 13, 2020). Available: https://www.iconfinder.com/icons/202809/redis_icon.

[126] Redis, "Redis FAQ," 2020 (accessed May 13, 2020). Available: https://redis.io/topics/faq.

[127] antirez, "Create-cluster script," 2020 (accessed May 13, 2020). Available: https://github.com/redis/redis/blob/a25df9dee0455ce3e072e6dd729736504bdad232/utils/create-cluster/create-cluster.

[128] Redis, "Redis cluster creation script," 2020 (accessed May 13, 2020). Available: https://github.com/antirez/redis/blob/94c026cd1971d1f0680c20f550e3094717d3b55e/utils/create-cluster/create-cluster.

[129] antirez, "HIREDIS-VIP GitHub issue 102," 2019 (accessed May 13, 2020). Available: https://github.com/vipshop/hiredis-vip/issues/102.

[130] H. Chu, "Sqlightning git," 2013 (accessed May 13, 2020). Available: https://github.com/LMDB/sqlightning.

[131] Postfix, "Postfix OpenLDAP LMDB howto," 2020 (accessed May 13, 2020). Available: `http://www.postfix.org/LMDB_README.html`.

[132] T. M. Project, "Monero git," 2020 (accessed May 13, 2020). Available: `https://github.com/monero-project/monero`.

[133] R. U. Würzburg, "High Performance Computing," 2018 (accessed May 13, 2020). Available: `https://www.rz.uni-wuerzburg.de/dienste/rzserver/high-performance-computing/`.

[134] Slurm, "Slurm website," 2020 (accessed May 13, 2020). Available: `https://slurm.schedmd.com/overview.html`.

[135] Zfsonlinux, "Zfsonlinux website," 2020 (accessed May 13, 2020). Available: `https://zfsonlinux.org/`.

[136] J. Barr, "EC2 high memory update – new 18 TB and 24 TB instances," 2019 (accessed May 13, 2020). Available: `https://aws.amazon.com/blogs/aws/ec2-high-memory-update-new-18-tb-and-24-tb-instances/`.

[137] Amazon, "Amazon EC2 dedicated hosts pricing," 2020 (accessed May 13, 2020). Available: `https://aws.amazon.com/ec2/dedicated-hosts/pricing/`.

[138] N. Provos and D. Mazières, "A future-adaptable password scheme," 1999 (accessed May 13, 2020). Available: `https://www.usenix.org/legacy/events/usenix99/provos/provos_html/node1.html`.

Appendix

A. Config for Cluster-Creation Redis

We use the *create-cluster* script [127] with the input *config.sh* from Listing 8.1 to create our Redis cluster.

```
BIN_PATH="../../src/"
CLUSTER_HOST=127.0.0.1
PORT=30000
TIMEOUT=2000
NODES=120
REPLICAS=0
PROTECTED_MODE=yes
ADDITIONAL_OPTIONS="--save 0 --maxmemory 4294967296\
 --maxmemory-policy noeviction"
```

Listing 8.1: Redis Cluster Configuration config.sh

B. B+-Tree Data of LMDB Evaluation

We give here a short overview of the internal structure of the databases' B+-Trees in Table B.1. This part is unique to LMDB, because the database is based on a tree structure – whereas Redis uses hash tables. We provide this data for an interested reader and to give the best insight of the tests.

The data is collected with the tool *mdb_stat*. Here one can see the growth of the depth and number of leaves of the internal B+-Tree. The table conforms our previous indication, that the growth of the database is also linear with internal tree values.

Table B.1.: Detailed Growth of LMDB

Number of Entries	Depth	Branch Pages	Leaf Pages	Overflow Pages
1k	2	1	22	0
10k	3	5	215	0
100k	3	35	2133	0
1m	4	399	21342	0
10m	4	4206	214007	0
100m	5	35630	2138976	0

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text.

Würzburg, 20. July 2020

.....
(Christoph Sendner)