Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Networked Systems and Services

Two Attacks on the TLSH Similarity Digest Scheme

Scientific Students' Association Report

Author:

Gábor Fuchs

Advisor:

Dr. Levente Buttyán
Roland Nagy

2023

Contents

Kivonat

A hasonlósági hash algoritmusok sok különböző téren használatosak (pl. digitális nyomozás, spam szűrés, malware detekció és malware klaszterezés), amelyek megkövetelik tőlük, hogy ellenállóak legyenek olyan támadások ellen, melyek képesek előállítani (A) szemantikailag hasonló bemeneteket, amelyek nagyon különböző hasonlósági hash-ekre képződnek le, vagy (B) teljesen különböző bemeneteket, amelyek nagyon hasonló hasonlósági hash-ekre képződnek le. Megmutatjuk, hogy a TLSH, egy széles körben elterjedt hasonlósági hash függvény, nem kellően robusztus az ilyen típusú támadások ellen, pontosabban bemutatunk egy-egy automatizált módszert futtatható szoftver binárisok funkcionalitásukat nem befolyásoló módosítására, amelyek elérik, hogy (A) a TLSH hash-ek alapján adott különbözőségi pontszám magas legyen az eredeti és a módosított bináris között, illetve (B) a módosított bináris TLSH hash-e nagyon hasonlítson egy tetszőlegesen megadható másik TLSH hash-re vagy teljesen megegyezzen azzal. Módszereinket értékeljük egy nagy malware adathalmaz mintáin, és megmutatjuk, hogy hatásosan alkalmazhatóak olyan minták előállítására, amelyeket nem detektál a SIMBIoTA, egy közelmúltban javasolt hasonlóság-alapú malware detekciós megoldás.

Abstract

Similarity Digest Schemes are used in various applications (e.g. digital forensics, spam filtering, malware detection and malware clustering), which require them to be resistant against attacks aiming at generating (A) semantically similar inputs with very different similarity digest values, or (B) completely different inputs with very similar digest values. We show that TLSH, a widely used similarity digest function, is not robust enough against either kinds of attacks, more specifically we propose automated methods to specially modify executable software binaries in a way that the modified binary has the exact same functionality as the original one, yet (A) its TLSH difference score from the original version becomes high, or (B) its TLSH digest becomes very similar to another arbitrary TLSH digest up to a complete hash collision. We evaluate our methods on a large data set containing malware binaries, and we also show that they can be used effectively to generate adversarial samples that evade detection by SIMBIoTA, a recently proposed similarity-based malware detection approach.

Chapter 1

Introduction

Similarity digest schemes map an input of arbitrary length to a small size output, such that similar inputs result in similar digest values. Such schemes are different from cryptographic hash functions that map even very similar inputs, differing only in a single bit, to completely different hash values. Similarity digest schemes, such as Ssdeep [9], Sdhash [18], Nilsimsa [7], and TLSH [13] are used in, for instance, digital forensics [3], spam filtering [5], malware clustering [20, 10], and malware detection [22]. In each of these applications, similarity of various files are measured based on the similarity of their digest values.

We differentiate two large classes of applications of similarity digest schemes: *fuzzy blacklisting* and *fuzzy whitelisting*.

A blacklist is a list of hashes of forbidden items (e.g., files, samples, *etc.*). When a blacklist is enforced, any item whose hash is on the blacklist is rejected, whereas everything else is accepted. The difference between fuzzy blacklisting and conventional blacklisting is that when fuzzy blacklisting is used, not only items with exact matches to the blacklisted ones are rejected, but also items that are similar to them. Use cases of fuzzy blacklisting include malware detection and spam filtering. In such use cases, what we expect from a similarity digest scheme is that it makes it rather difficult for an attacker to generate semantically similar inputs that have very different similarity digest values. If that was easily possible, then attackers could defeat similarity-based malware detection and spam filtering in an *anti-blacklisting* attack by modifying otherwise forbidden files in a way that their digests become very different from those of the original versions. Such an attack would also effect many other use cases, for example, file similarity analysis in forensic investigations and clustering malware samples based on their similarity digests would make no sense.

A whitelist is a list of hashes of acceptable items. When enforcing a whitelist, every item whose hash is on the whitelist is accepted, while everything else is rejected. Fuzzy whitelisting differs from traditional whitelisting by not only accepting exact matches to the acceptable items, but approximate matches as well. Example applications include fuzzy whitelisting of binaries [17] and network traffic [4]. In this case, what we expect from the similarity digest scheme is that it makes it difficult to create completely different inputs that have very similar or identical similarity digest values. An *anti-whitelisting* attack has complete control over the similarity digest of the adversarial file by making only limited modification to the file, and modifies the digest to be similar to one of the digests on the whitelist. An ideal version of this attack creates a perfect hash collision. With a successful attack like that, attackers could defeat not only fuzzy whitelisting applications, but essentially **all** uses of the similarity digests.

Robustness of existing similarity digest schemes against such attacks have been extensively studied [2, 19, 11, 14], and the authors of [14] concluded that Ssdeep and Sdhash are not sufficiently robust for practical use, while TLSH is more difficult to exploit.

In this paper, we show that it is rather easy to exploit TLSH too. More specifically, we propose both an anti-blacklisting and an anti-whitelisting attack against the TLSH scheme. Our methods modify executable files (binaries), such that the modified binary has the exact same functionality as the original one, while their similarity digest values are very different (anti-blacklisting), or the similarity digest of the modified binary is very similar to a desired arbitrary target digest (anti-whitelisting). Our methods do not involve any encryption or packing techniques: they preserve the original text and data segments of the binary, hence, besides remaining semantically equivalent, the modified file also remains syntactically similar to the original one. In our anti-blacklisting attack we try to maximize the difference score returned by the official TLSH difference calculation function for the digests of the modified and the original binaries by modifications that are also very limited in size. As a practical application of the method, we also show how it can be used for generating adversarial malware samples that evade SIMBIoTA, a recently proposed similarity-based malware detection mechanism for embedded IoT devices [22]. In the anti-whitelisting attack, we achieve a complete collision of the modified binary's TLSH hash to a given arbitrary TLSH hash. We demonstrate how such an attack can be used to create adversarial malware samples from any existing malware sample such that the adversarial samples have the exact same TLSH hash as a well-known benign binary. This makes it impossible to correctly classify them as malware or differentiate them in any other way from the benign binary (or each other) based on their TLSH hashes alone. Finally, we note that our methods can be adapted to other types of inputs (e.g., images and documents) where the file format allows for the modification of some parts of the file without affecting its semantics and corrupting its formatting rules.

The paper is organized as follows: In Chapter 2, we introduce the necessary background on the operation of the TLSH similarity digest scheme, we discuss its robustness analysis presented in [14], and we introduce some details of the ELF file format. In Chapter 3, we discuss our anti-blacklisting attack: Section 3.1 presents the design of our anti-blacklisting method for modifying ELF binaries such that the functionality of the modified binary remains the same, its appearance remains similar to the original binary, yet the TLSH difference score between the original and the modified binaries becomes high. In Section 3.2, we evaluate our method on a large data set containing malware binaries, and we study how effectively it can be used to evade similarity-based malware detection, as well as how it could be adapted to other configurations and applications of TLSH. In Chapter 4, we discuss our anti-whitelisting attack: In Section 4.1, we present the design of the anti-whitelisting attack explaining how the value of each of the TLSH hash fields can be controlled to match the value of the corresponding field of the target hash, and thus create the desired hash collision. Section 4.2 reports on how we used the attack to generate adversarial malware samples that have the exact same TLSH hash as a well-known benign executable from samples of the same dataset as the one used in Chapter 3. Finally, in Chapter 5, we conclude our paper.

Chapter 2

Background

TLSH is a similarity digest scheme. As such, it provides two algorithms, the first of which calculates the fixed size 35 byte TLSH hash of a file, which preserves its characteristics in a way that the second algorithm can quantify how different the two files are based on their corresponding TLSH hashes. In terms of design the closest ancestor of TLSH among the well adopted similarity schemes is Nilsimsa, another locality sensitive hash. TLSH can be perceived as its much advanced, more complex, and fine-tuned version. In the analysis of Nilsimsa [7] multiple adversarial possibilities where shown against the scheme, the final one of which was a targeted (aimed) attack that entailed precise and efficient manipulation of the scheme utilizing deeper understanding of the algorithm. As our attacks against TLSH are similar in nature, they require detailed discussion of the TLSH algorithm.

2.1 The TLSH algorithm

In this section we discuss the TLSH algorithm. Note that its presentation in [13] is somewhat confusing regarding details like byte order. The following is based on the official reference implementation[1]. TLSH has a few alterable parameters; however, they all have default values, which are carefully chosen [12], and TLSH is mostly used with these defaults. Hashes calculated with different parameters are incompatible for difference calculation. For these reasons we only discuss TLSH with default parameters.

The TLSH algorithm takes groups of 3 bytes (*byte triads*) as input features, which are selected the following way. The file is traversed with a 5 byte sliding window stepped byte-by-byte. For each window position the algorithm takes all possible selections of three bytes that where not contained in previous window positions (i.e. the ones containing the last byte of the window). For each window position these are 6 new byte triads. With the terminology of k-skip-n-grams[2], from all window positions together they are the 2-skip-3-grams ($n = 3$, $k = 2$) of the whole processed byte stream[3]. Each of these byte triads is hashed to a single byte (Figure 1). The occurrences of different hash values (throughout all the processed byte triads in all the window positions) are counted in a zero initialized 256 element array. With the analogy of bucket hashing, the elements of the array are called *bucket counts*. As it is possible that more than one of the 6 hash values are the same, a bucket count can be increased by more than one over a single position of the sliding window. The most extreme examples of this, which we have found by exhaustive search,

[1] `https://github.com/trendmicro/tlsh` Last accessed: June 26, 2023

[2] Groups of n almost consecutive bytes with skipping at most k bytes in total in between.

[3] Excluding the ones that could be formed using the first 4 bytes of the file.

are the window contents `533df60525`, `212be01325`, `95f932c125`, and `4aef24d725` as each of them increases a single bucket count (for hash values 228, 210, 9, and 47 respectively) by 6. Along with increasing the bucket counts, a single-byte checksum is also calculated (Figure 2.1).

Default TLSH preserves the bucket counts only for hash values 0-127, discarding the second half of the array.

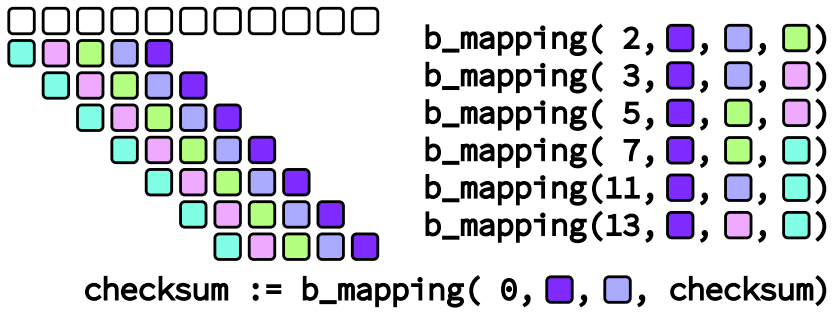

Figure 2.1: The TLSH sliding window and the calculations with the selected bytes: the formula for the 6 single-byte hash values, and the update of the checksum value for each window position. `b_mapping` is a deterministic hash function based on the Pearson hash [16].

After traversing the file for the bucket counts and the checksum, next the quartiles q_1, q_2, and q_3 of the kept 128 bucket counts are calculated. If we sort the bucket counts in ascending order, q_1, q_2, and q_3 will be the values of the 32nd, 64th, and 96th ones respectively, in other words, the last in each of the first three quarters.

The TLSH hash itself consists of the following values:

- `checksum`: the one byte checksum;
- `lvalue`: the length of the file on a logarithmic scale in one byte;
- `Q1ratio` and `Q2ratio`: two half bytes for the ratios of the quartile ratios in percentage modulo 16:

  $\texttt{Q1ratio} = \lfloor 100\frac{q_1}{q_3} \rfloor \bmod 16$ and $\texttt{Q2ratio} = \lfloor 100\frac{q_2}{q_3} \rfloor \bmod 16$;
- `codes`: 128 quarter bytes for relations of each bucket count to the quartiles in the original order:

$$\texttt{code}_i = \begin{cases} 0 = \texttt{00} & \text{if } b_i \leq q_1 \\ 1 = \texttt{01} & \text{if } q_1 < b_i \leq q_2 \\ 2 = \texttt{10} & \text{if } q_2 < b_i \leq q_3 \\ 3 = \texttt{11} & \text{if } q_3 < b_i \end{cases}$$

For easy reference, we refer to each of them in the same way as the already mentioned official TLSH implementation does.

TLSH does not provide hashes for some low-entropy data, like repetitions of a short pattern. This could cause only a few bucket counts being increased and most of them left zero, leading to all quartiles being zero too. In this case, if we tried calculating the hash header values, specifically the quartile ratios, we would have to divide by zero.

The difference between two TLSH hash values is calculated from the relations of the above described values in the two compared hashes. Each field has its own possible contribution to the difference, and these contributions are simply summed. To compare fields that are stored modulo-N, the `mod_diff` method is introduced. $\texttt{mod\_diff}(x, y, R)$ is the smallest non-negative integer congruent to either $x - y$ or $y - x$ modulo R. The contributions of the various TLSH fields to the total difference are the following (subscripts $_a$ and $_b$ mark fields of the two compared TLSH hashes):

- `checksum`: $\texttt{diff}_{\texttt{checksum}} = \begin{cases} 0 & \text{if } \texttt{checksum}_a = \texttt{checksum}_b \\ 1 & \text{if } \texttt{checksum}_a \neq \texttt{checksum}_b \end{cases}$
- `lvalue`: having $\texttt{ldiff} = \texttt{mod\_diff}(\texttt{lvalue}_a, \texttt{lvalue}_b, 256)$;
  (the use of `mod_diff` is interesting here[4])
  $\texttt{diff}_{\texttt{lvalue}} = \begin{cases} \texttt{ldiff} & \text{if } \texttt{ldiff} \leq 1 \\ 12 \cdot \texttt{ldiff} & \text{if } \texttt{ldiff} > 1 \end{cases}$
- `Q1ratio`: having $\texttt{q1diff} = \texttt{mod\_diff}(\texttt{Q1ratio}_a, \texttt{Q1ratio}_b, 16)$,
  $\texttt{diff}_{\texttt{Q1ratio}} = \begin{cases} \texttt{ldiff} & \text{if } \texttt{q1diff} \leq 1 \\ 12 \cdot (\texttt{q1diff} - 1) & \text{if } \texttt{q1diff} > 1 \end{cases}$
- `Q2ratio`: having $\texttt{q2diff} = \texttt{mod\_diff}(\texttt{Q2ratio}_a, \texttt{Q2ratio}_b, 16)$,
  $\texttt{diff}_{\texttt{Q2ratio}} = \begin{cases} \texttt{ldiff} & \text{if } \texttt{q2diff} \leq 1 \\ 12 \cdot (\texttt{q2diff} - 1) & \text{if } \texttt{q2diff} > 1 \end{cases}$
- `codes`: having $d_i = |\texttt{code}_{i,a} - \texttt{code}_{i,b}|$,
  $\texttt{diff}_{\texttt{codes}} = \sum_{i=0}^{127} \begin{cases} d_i & \text{if } d_i \in \{0, 1, 2\} \\ 6 & \text{if } d_i = 3 \end{cases}$

The non-linear scoring of `ldiff`, `q1diff`, `q2diff`, and each d_i are approximations of functions defined by (probabilistic) considerations of the effects of random modification.

2.2 Robustness of TLSH

Robustness of Ssdeep, Sdhash and TLSH was compared in [14]. This entailed tests on spam images, texts, and web pages, as well as executable files. Here, we review some key features and conclusions of their tests and results on executable files.

In [14], the similarity detection threshold of TLSH difference was tuned based on the resulting false positive rate of tagging pairs of different executable files similar. The executable files used in this step were executable binaries from a Linux distribution. Based on the results, they stated that if an executable could be modified without altering its functionality in a way that the TLSH difference of the original and modified versions are at least 86, they would consider the digest scheme broken (see section 5.1 of [14]).

They conducted different random modifications in the source codes of some programs without altering their functionality, and compiled them with the modifications. To name

[4] `lvalue` has a maximal value 169. `mod_diff` is most likely chosen here instead of $|\texttt{lvalue}_a - \texttt{lvalue}_b|$, to limit $\texttt{diff}_{\texttt{lvalue}}$ to $12 \cdot 128$; however, it results in `lvalue` 0 and 169 ($\texttt{ldiff} = 87$) being scored much less different than `lvalue` 0 and 128 ($\texttt{ldiff} = 128$).

a few of their choices of modifications: reordering operands of commutative logical operations, introducing new variables, changing the order of function definitions, adding NOP instructions, or adding random binary data in character arrays. With extents of such random modifications that caused Ssdeep and Sdhash to mark the files completely different, TLSH still showed that they were related. One of the highlighted advantages of TLSH is that it does not have a concept of completely different files, as TLSH does not have a hard maximum of difference score[5], while Ssdeep and Sdhash score similarity on a 0-100 scale, and are likely to give the score 0 to a pair of unrelated files.

2.3 Modifying ELF binaries

We demonstrate our attacks against the TLSH scheme on the example of IoT malware samples, so the files we want to modify are ELF binaries, more specifically ELF executables. These files contain the instructions of the program and the data required during its execution. Modifying these in a previously unknown binary without changing the behavior of the program would be a very hard task. Fortunately these files have parts that are not essential to their execution, and thus, are promising candidates for modification.

As most file formats, ELF files can have multiple different parts, which are addressed in headers. The first header is the ELF header. Every ELF file must have this, and it must be right at the beginning of the file. As there are multiple types of ELF files (executables being only one of them) and ELF files support countless system architectures, the ELF header contains values that specify what the current file is in terms of these possibilities. One of the first few of these is the `EI_CLASS` byte, which specifies whether the current file is for a 32-bit or a 64-bit architecture. The value of this byte is important, because the structure of the ELF header itself is different for these two cases. The reason of this difference is the presence of memory-offset-like values in the header, which take up the corresponding amount of bytes to the designation of each architecture: 4 bytes for 32-bit and 8 bytes for 64-bit. The ELF header has three fields of such type: `e_entry`, `e_phoff`, and `e_shoff`. Generally all of these three are optional as not all kinds of ELF binaries need them. If the file does not use one of them, it will have the value 0.

`e_entry` is required in executable files as this specifies the starting point of the execution of the instructions. `e_phoff` marks the start of the so called *program header table*, while `e_shoff` marks the start of the *section header table*.

Both the program header table and the section header table are called *header tables*, because they both are arrays of equal-sized headers called *program headers* and *section headers*, respectively. Program headers describe how *segments* of the file are mapped to memory ranges when loading the binary for execution. These are also required in executable files. Section headers, on the other hand, describe parts of the file itself, and usually are not used at all for the execution of the file. The size of these header types and the number of headers in each table are also usually specified in the ELF header. The only exceptions to this are when the number of headers in one or both of the tables is very high[6], in which case that count is stored in fields of the first special element of the section header table.

[5] Technically there is maximum possible TLSH difference score as the TLSH difference is a sum of a fixed number of smaller differences that all have their maximum values. The sum of these individual maximums is 2473 points of total difference. While this value is not achievable as a difference of two TLSH hashes, the real maximal value is definitely smaller.

[6] `0xffff` or larger for the program header table and `0xff00` or larger for the section header table

The section header table can contain references to many parts of the file that could be overwritten without any fear of damaging the behavior of the program, but for simplicity we target only the section header table itself (in the first attack – Chapter 3). This is done by calculating the start and end of the table in the file, and then making sure that no one is reading the data we corrupt with our modifications. This is achieved by removing the reference to the table by changing the ELF header values `e_shoff` and `e_shnum` to 0. The latter is the field that stores the number of entries in the section header table, which must be 0 if there is no section header table at all in the file.

Another simple possibility to add additional content to an ELF file without affecting its behavior is to simply append to the end of the file (used in the second attack – Chapter 4). During execution and other use of the file the operating system reads the file headers and finds other required parts of the file based on absolute references from either the start of the file or in the memory after loading, or relative references. The size of the file is not hardcoded anywhere and there are no references that are relative to the end of the file.

Chapter 3

Attack I: anti-blacklisting

Making the similar look unsimilar.

In this chapter we present and evaluate our first attack, which aims to modify arbitrary executable ELF binaries in such a way that

1. their functionality remains unaffected;
2. their size remains unchanged; and
3. the TLSH difference of their modified and original versions (*self difference*) becomes as high as possible.

Requirement 2 serves to rule out the exploitation of a known limitation of TLSH, namely the fact that it can be easily manipulated by appending large amounts of random data or the bytes of benign software (see for example [21]). While this is a weakness in this case, it is by design and needs to be dealt with in applications like malware detection. We would like to manipulate TLSH in a more sophisticated way without adding much unused content to the file. While not changing the file size at all is definitely overkill, it is a great challenge to test the space efficiency of our method, which might be crucial for a stealth attack against possible more advanced detection methods.

3.1 Design

3.1.1 Manipulating the TLSH difference

The TLSH hash is almost entirely defined by the bucket counts. As these are simply summarized over the whole file, a trivial way of manipulating the TLSH hash of a file consists in appending huge amounts of data to it. For example, if we take some large chunk of data that affects enough bucket counts, then appending this chunk of data to the file again and again would result in the ratios of the bucket counts of the modified file converging to the bucket count ratios of the periodically appended data.

However, our goal is to manipulate the TLSH hash, and thus the TLSH difference of the modified and the original versions of the file, by modifying only a limited portion of the file and keeping the file size unchanged. As every part of the TLSH hash affects the TLSH difference, we have a few choices to achieve the desired difference by our modifications.

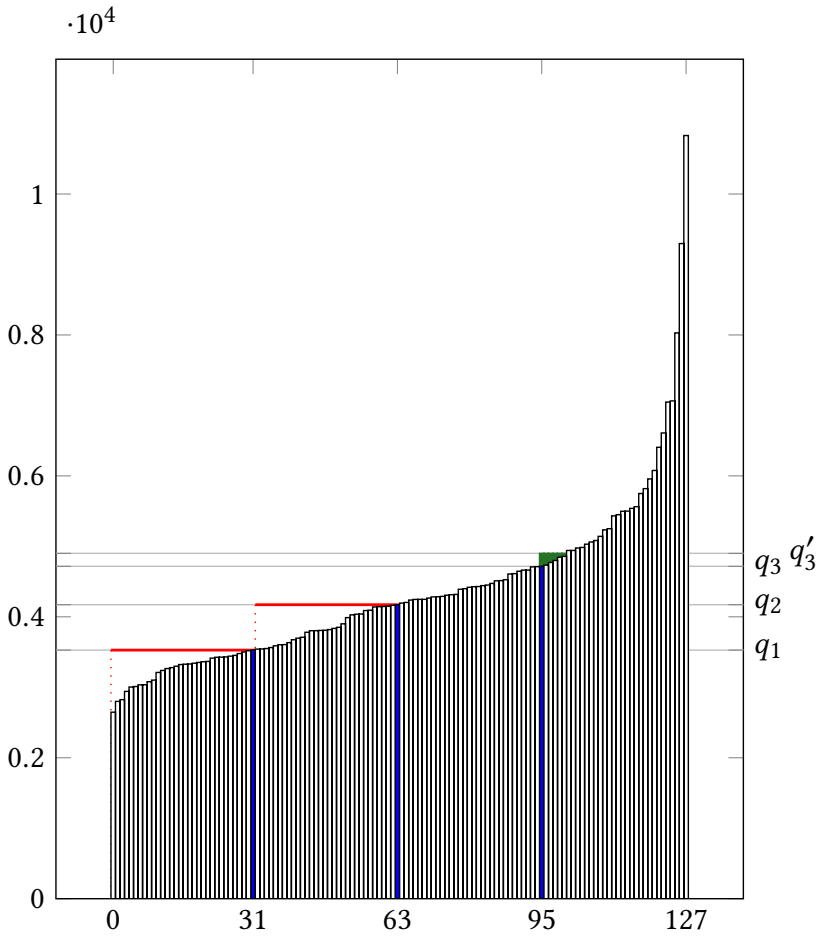

Figure 3.1: The sorted bucket counts for one of the tested malware samples. Highlighted are the quartile defining counts, the described minimal required changes to increase q_3 to the target value, and the bucket count limits that must not be exceeded not to increase q_1 or q_2.

The `checksum` is basically irrelevant as it can provide only a single point of difference. The `lvalue` likewise, because we do not want to change the size of the file. This leaves us the `Q1ratio`, `Q2ratio` and the `codes`.

Our chosen strategy is the following: Let's try to increase q_3 to $q_3' > q_3$ without changing q_1 or q_2 to have both $r_1 = \lfloor 100\frac{q_1}{q_3} \rfloor$ and $r_2 = \lfloor 100\frac{q_2}{q_3} \rfloor$ decrease by the integer numbers d_1 and d_2 (i.e., $r_i' = r_i - d_i$), such that $2 \leq d_1, d_2 \leq 8$ and $(d_1 - 1) + (d_2 - 1) \geq \lceil \frac{t}{12} \rceil$, where t is the target self difference score to achieve. This would result in at least $12 \cdot \lceil \frac{t}{12} \rceil \geq t$ difference in total from the original version. For r_i to decrease by d_i, the new q_3' value needs to be at least $\lfloor \frac{100 \cdot q_i}{r_i - (d_i - 1)} \rfloor + 1$. We choose the minimal q_3' target value that provides the above difference in this way. For example, for $32 < t \leq 48$, this strategy usually provides $d_1 = d_2 = 3$.

If we had a way of precisely increasing a few selected bucket counts, this could allow us to increase q_3 to q_3' by increasing the bucket count corresponding to q_3, as well as the few following bucket counts in the ascending order that are still below the desired q_3' value (see Figure 3.1). To be able to precisely modify a single bucket count or just a few at a time like this, we have to learn how different byte-patterns written to the file affect the bucket counts.

3.1.2 Patterns

As the space for modification is limited, the challenge is not just to find an array of bytes that, when inserted to the file, changes the bucket counts in the desired way, but to find one that also fits into this limited space. We can say that the cost of each pattern we decide to use is its length.

3.1.2.1 Periodic patterns

Whenever the sliding window selects the same 5 bytes, the same 6 values are calculated, and so the same bucket counts increase by the same amounts. If we want to significantly increase a few bucket counts without changing others, which is indeed needed for modifications like the one visualized in Figure 3.1, periodic patterns can be very efficient, as they cause a few fives of byte values to be selected, and thus the same bucket counts increased again and again.

As an example, let's see what bucket counts increase when the sliding window traverses the periodic ASCII encoded text `lalalala`:

1. The first five bytes selected are the characters `lalal`. The six hash values calculated from them are 155, 195, 63, 100, 193, and 43. As each of these are different, each of the corresponding bucket counts are increased by 1.
2. The next selection is `alala`, which hashes to 104, 43, 119, 41, 10, and 105.
3. The next selection is `lalal` again, which again hashes to 155, 195, 63, 100, 193, and 43.
4. Finally, the last selection is `alala` again, which again hashes to 104, 43, 119, 41, 10, and 105.

If we discard the bucket counts for the values ≥ 128, as default TLSH does, we get the following total increments from this: 10: +2, 41: +2, 43: +4, 63: +2, 100: +2, 104: +2, 105: +2, 119: +2. The count for bucket 43 is increased by 4 as all four steps increased it by 1. In other cases, as stated earlier, it is also possible that a count is increased by more than 1 in a single step.

These 4 steps in total were over 2 iterations of a pattern with period length 2. If we continue, and add more iterations by appending the bytes `la` more times to the end, the total count increments for each affected bucket will scale linearly with the number of iterations.

For a pattern with period length p, adding an extra iteration only costs additional p bytes; however, the first iteration costs more as it takes at least five bytes to fill the sliding window. Hence, the total length (cost) of a periodic pattern is $4 + n \cdot p$ bytes, where n is the number of iterations that all linearly provide the same increments in the bucket count contributions. So, for example, with the repetition of the characters `la` from the above example, the total pattern that takes up $4 + n \cdot 2$ bytes would increase the bucket counts by: 10: $+n$, 41: $+n$, 43: $+2n$, 63: $+n$, 100: $+n$, 104: $+n$, 105: $+n$, 119: $+n$.

3.1.2.2 Neutral periodic patterns

There is an interesting consequence of TLSH discarding bucket counts 128-255: there are arrays of bytes which do not increase the kept first 128 bucket counts at all. This is because the hashes of all of their processed byte triads are ≥ 128.

Merging this with the idea of periodic patterns we can find what we call *neutral periodic patterns*: periodic patterns that do not increase the kept 128 bucket counts. While there are many examples of such neutral patterns, we use the easy to remember ASCII string `YYYYY`. With its period length one, we can fill out any ranges of bytes, thus erasing their contributions to the kept bucket counts.

3.1.2.3 The pattern database

We know what bucket counts to increase, and that there might be some periodic patterns that can achieve this efficiently in terms of used space in bytes without affecting many other bucket counts. Next, we just have to find the appropriate patterns.

We take the trivial approach and try every possible periodic pattern up to a limited period length. There are 256 different patterns with period length $p = 1$, 256^2 with $p = 2$, and generally 256^p patterns with period length p.

In our experiments we tried using period lengths up to 3, but only used 1 and 2 in the final setup, as these proved to be sufficient to efficiently manipulate most required combinations of bucket counts. These are 256^2 patterns in total as the ones with $p = 1$ are all included in the set of patterns with $p = 2$.

For this data to be quickly available, we created files containing the single iteration bucket count contributions of each of these patterns. Another set of bucket contributions is also stored for each pattern, which is discussed in Section 3.1.3.4. We stored data for both $p = 1$ and $p = 2$ patterns, as $p = 1$ patterns can be easily scaled to an odd number of bytes as well.

3.1.3 The steps of patching

Equipped with all the ideas described above, the steps of modifying ELF binaries are the following:

3.1.3.1 Load the binary

The first step is to load the binary. As we have to achieve the TLSH difference from its original version, we calculate r_1 and r_2[1] from the data before any modifications. Let's call these $r_{1,\text{orig}}$ and $r_{2,\text{orig}}$.

3.1.3.2 Take the range

Now we can take the section header table, and we will have the range that we can freely modify. This is done as described in Section 2.3.

[1]Note that these are not the final `Q1ratio` and `Q2ratio` values. `Q1ratio` $= r_1 \bmod 16$ and `Q2ratio` $= r_2 \bmod 16$

Note that if we write a given periodic pattern in this range, we increase some bucket counts in a predictable manner, but at the same time, some bucket counts would also decrease, as we also remove data by overwriting them with the chosen periodic pattern. To address this uncertainty, the next step is to overwrite the available range with the neutral periodic pattern `YYYYY`. This is expected to change many bucket counts; they mostly decrease, but technically it might increase a few along the edges of the range, where the `Y` bytes meet the last 4 and first 4 bytes before and after the range. Neither of these are problematic, as this will be the new reference state of the whole file, and the starting point for calculative modifications. We calculate the starting bucket count quartiles q_1, q_2, and q_3 now.

3.1.3.3 Calculate target

Now we are ready to calculate the target q_3' value to achieve a self difference score $\geq t$, which we do as described in Section 3.1.1. The only difference is that we have to achieve the difference from the original values $r_{1,\text{orig}}$ and $r_{2,\text{orig}}$, not the current values that could be calculated from q_1, q_2, and q_3. We are looking for the minimal q_3' target value that satisfies Inequality 3.3 and Inequality 3.6. This can be done by trying each value increasingly, beginning from the current q_3 value.

$$r_i' = \lfloor 100 \cdot \frac{q_i}{q_3'} \rfloor \qquad i = 1, 2 \tag{3.1}$$

$$d_i = r_{i,\text{orig}} - r_i' \qquad i = 1, 2 \tag{3.2}$$

$$2 \leq d_i \leq 8 \qquad i = 1, 2 \tag{3.3}$$

$$\texttt{diff}_{\texttt{Q1ratio}} = (d_1 - 1) \cdot 12 \tag{3.4}$$

$$\texttt{diff}_{\texttt{Q2ratio}} = (d_2 - 1) \cdot 12 \tag{3.5}$$

$$\texttt{diff}_{\texttt{Q1ratio}} + \texttt{diff}_{\texttt{Q2ratio}} \geq t \tag{3.6}$$

Now that we have q_3', we calculate the required bucket count increments exactly as explained in Section 3.1.1 and visualized in Figure 3.1.

3.1.3.4 Choose and fill the patterns – Strategy

At this point we have a byte range currently filled with the neutral pattern of `Y` bytes, to which we can write anything. We know exactly which bucket counts to increase to what values to increase q_3 to the desired q_3' value; and which bucket values we must not increase above certain limits (not to increment q_1 or q_2 by doing so). We also have 256^2 different periodic patterns, which we can use to space (cost) efficiently increase bucket counts.

If we use a periodic pattern, its periodic bucket count contributions scale linearly with the number of its iterations. There is another contribution, which we have not discussed to this point yet. This is the pattern's *side contributions*, which are the bucket count increments it causes in the sliding window positions at the beginning and at the end when the window is only partially filled with the bytes belonging to the pattern (i.e., when the window also contains some of the bytes surrounding the periodic pattern). As the sliding window always selects 5 consecutive bytes, this interaction in the bucket count contribution is only to the last and first 4 bytes before and after the pattern.

To be able to pre-calculate this side contribution of each pattern and store it in the pattern database as well, we always surround each used pattern with `YYYY` (i.e., at least 4 bytes of our neutral periodic pattern) on each side. This means that the complete modifiable

range, which is currently filled with `Y` bytes, will still have ranges of at least 4 such bytes after all of our modifications at the beginning, at the end, and in between separating the used patterns from each other and from the bytes surrounding the whole modifiable range.

The choice of using `Y` bytes for this purpose too has two reasons. One is that sliding window positions ending in at least 3 `Y` bytes have some guaranteed neutral (128-255) hashes, which makes the expected bucket count increments for passing these in between arrays of bytes a little lighter (this is actually only $\binom{4}{3} = 4$ out of $8 \cdot 6 = 48$ hashes), which is favorable, since we use periodic patterns for their periodic contributions. While side contributions can be lucky sometimes, they are harder to control, and sometimes harmful, as they can make good patterns unusable by increasing bucket counts that we must not.

The other, much more significant reason is that this way we do not have to fill the whole modifiable range if there is a solution that requires fewer bytes of the selected patterns; because the trailing `YYYY` after the last pattern will blend with the remaining neutral pattern at the end of the range.

3.1.3.5 Choose and fill the patterns – Problem

Now the task of choosing patterns to write to the range in a way that the desired bucket count increments and limitations are fulfilled comes down to solving inequalities (3.7) and (3.8) for n_j.

In the sequel, we use the following notation:

- L is the length of the whole modifiable range,
- p_j is the period length of pattern j,
- n_j is the number of its used iterations,
- $c^P_{j,i}$ is its primary/periodic contribution to bucket count i,
- $c^S_{j,i}$ is its secondary/side contribution to the same bucket count,
- t_i is the target increment for bucket count i, and
- l_i is the increment limit for bucket count i.

For the bucket counts that have no target increment, t_i is 0, and for bucket counts that are not limited, l_i is ∞.

$$t_i \le \sum_j \left(n_j c^P_{j,i} + \begin{cases} 0 & \text{if } n_j = 0 \\ c^S_{j_i} & \text{if } n_j > 0 \end{cases} \right) \le l_i \tag{3.7}$$

$$4 + \sum_j \left(n_j p_j + \begin{cases} 0 & \text{if } n_j = 0 \\ 4 + 4 & \text{if } n_j > 0 \end{cases} \right) \le L \tag{3.8}$$

Inequality (3.7) is about the increments of each bucket count. The increment has to be at least t_i and at most l_i to achieve the goal of increasing q_3 to q'_3 without increasing q_1 or q_2. The increment is the sum of the total periodic and side increments for each pattern. The total periodic increment for a pattern is $n_j c^P_{j,i}$, a single iteration increment times the number of iterations used; while the side increment is always $c^S_{j_i}$, but of course it is only there if we choose to use at least one iteration of the pattern.

Inequality (3.8) is the requirement of the selected patterns fitting into the whole modifiable range. So the complete number of bytes used must be at most L. The modifiable range starts with `YYYY`, which is 4 bytes; then there is $4 + n_j p_j$ bytes with another 4 bytes of trailing `YYYY` for each pattern used. The formula is structurally very similar to the one in inequality (3.7), because just as there is a set of constant bucket count contributions for each used pattern regardless of the number of iterations, there is also a constant amount of bytes used.

3.1.3.6 Choose and fill the patterns – Solution

To solve the system of inequalities (3.7) and (3.8), we use a solver. Solvers are great, but can take ages to solve a hard problem with lots of variables. In this problem the variables are the n_j numbers of used iterations of all patterns, which are 256^2 variables.

To decrease the number of these variables, we select the "best" few[2] patterns for the current problem based on some heuristics: we give each pattern a score that attempts to quantify how useful its contributions are to solve the concrete current problem.

The scoring rules are simple:

1. If the pattern has any periodic contribution to any limited bucket count, we disqualify it by assigning the score 0.
2. If the pattern has a side contribution that increases a limited bucket count over its limit by itself, we also disqualify it by assigning the score 0.
3. Otherwise, each pattern is scored to the sum of its increments for each bucket count with a target value divided by its period length.

Note that the applicability of rule 1 is heavily dependent on our chosen strategy to increase q_3 specifically, and thus there are no bucket counts that have both a target value and a limit (see Figure 3.1).

So we find the few patterns with the highest scores, and we use a solver to solve the system of inequalities (3.7) and (3.8), considering them to be all the available patterns. If we got a solution, we compile it to a patch on the whole modifiable range as described in Section 3.1.3.4.

The version of the binary patched in this way

- should work just like the original version of the executable,
- has the same size as the original file,
- is guaranteed to have a self difference larger than or equal to the desired target value t.

[2] 32 or 16 in our experiments

3.2 Evaluation

3.2.1 Data set

The data set used for the evaluation of our attack setup is the *CrySyS-Ukatemi benchmark dataset of IoT malware 2021* or CUBE-MALIoT-2021[3]. This data set is publicly available for research and education purposes, and consists of 29,209 ARM and 18,715 MIPS malicious ELF binaries (also called malware samples).

3.2.2 Results of patching

We tested our attack on 2000 randomly selected ARM malware samples from CUBE-MALIoT-2021. We tried a few different configurations of the attack with various self difference target values, trying more periodic patterns including the ones with a period length 3, and preselecting different amounts of patterns for the solver to work with. In our final experiment, we tried to adaptively patch for guaranteed self differences in steps of 12 points, which is the increment achievable with each additional percent of change in any of the quartile ratios. Each increment is tried if the previous one succeeded, until one fails or the maximal difference is reached. We used all the patterns with period lengths 1 and 2, and preselected the best 16 patterns for the solver to work with in each patching attempt.

A failed attempt to patch might fall into four categories by different reasons:

- No SHT – there was no section header table in the binary, which is the only part that our setup currently supports modifying;
- Bad patterns – the patterns chosen by our heuristics were insufficient to solve the problem;
- Short ranges – the section header table was too short to solve the problem with the selected patterns;
- Timed out – the solver hung for a long time without finding a solution or proving that there is none.

As the section header table is not required in executable files, it was already missing in 221 malware samples out of the 2000 samples that we tested. Our setup does not support the modification of these files (*No SHT*). The remaining 1779 samples contain section header tables, with the smallest in size of these tables being 120 bytes, the largest being 1560 bytes, and the average size being 774.4 bytes.

Bad patterns and *Short ranges* mark the solver failing at two different steps. We do not include separate failing rates on one or the other of these reasons as not determining the cause of solver failure proved to speed up the attack significantly. In our earlier experiments we distinguished the two separate failure causes to evaluate our heuristics to preselect the patterns, and based on the small relative frequency of *Bad patterns*, we concluded that the method is good enough.

Timed out was frequent in experiments with high target difference settings and a larger number of patterns passed to the solver to work with. This cause of failure was completely eliminated by reducing this number of passed patterns to the current 16. As this change

[3] `https://github.com/CrySyS/CUBE-MALIoT-2021` Last accessed: June 27, 2023

reduced the success rate on lower target difference settings where *Timed out* was not a problem, further optimization is possible.

In Figure 3.2, the number of successfully patched binaries is presented for each increment of target difference. The actual achieved differences on the highest successful setting for each sample are visualized in Figure 3.3.

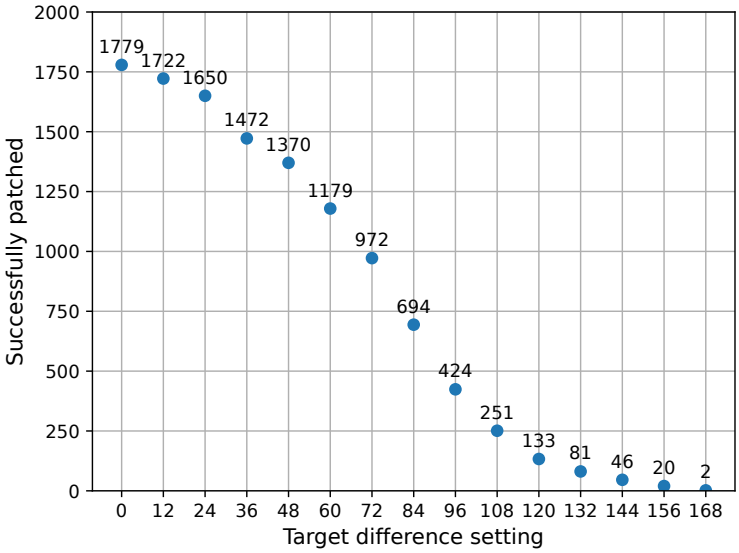

Figure 3.2: Results on 2000 ARM samples. Each value represents how many samples were successfully patched on the given or a higher setting. Setting 0 refers to the number of all samples that had a section header table to work with.

3.2.3 Comparison to other research

As reviewed in Section 2.2, the robustness of TLSH (along with other similarity hashes) was tested in [14] on a few kinds of data including executable files.

Their goal, just as ours is to modify an executable binary without changing its functionality, in a way that the original and the modified version have a high TLSH difference. However, there are some key differences between their and our current approach:

- they modify source code and compile it again, while we modify the binary directly without having access to the source code;
- they consider changing and reordering functional elements without actual change in functionality, while we attack by removing and overwriting an unnecessary part of the binary itself, without even touching functional parts, like the text or data segments;
- their modifications are random, and their purpose is to equally challenge multiple different similarity schemes, while our modifications are very targeted and try to exploit certain weaknesses of TLSH hash and difference calculation.

Their proposed similarity detection threshold for executables is the TLSH score of 86; and in the context of their research, they stated that they consider the digest scheme broken

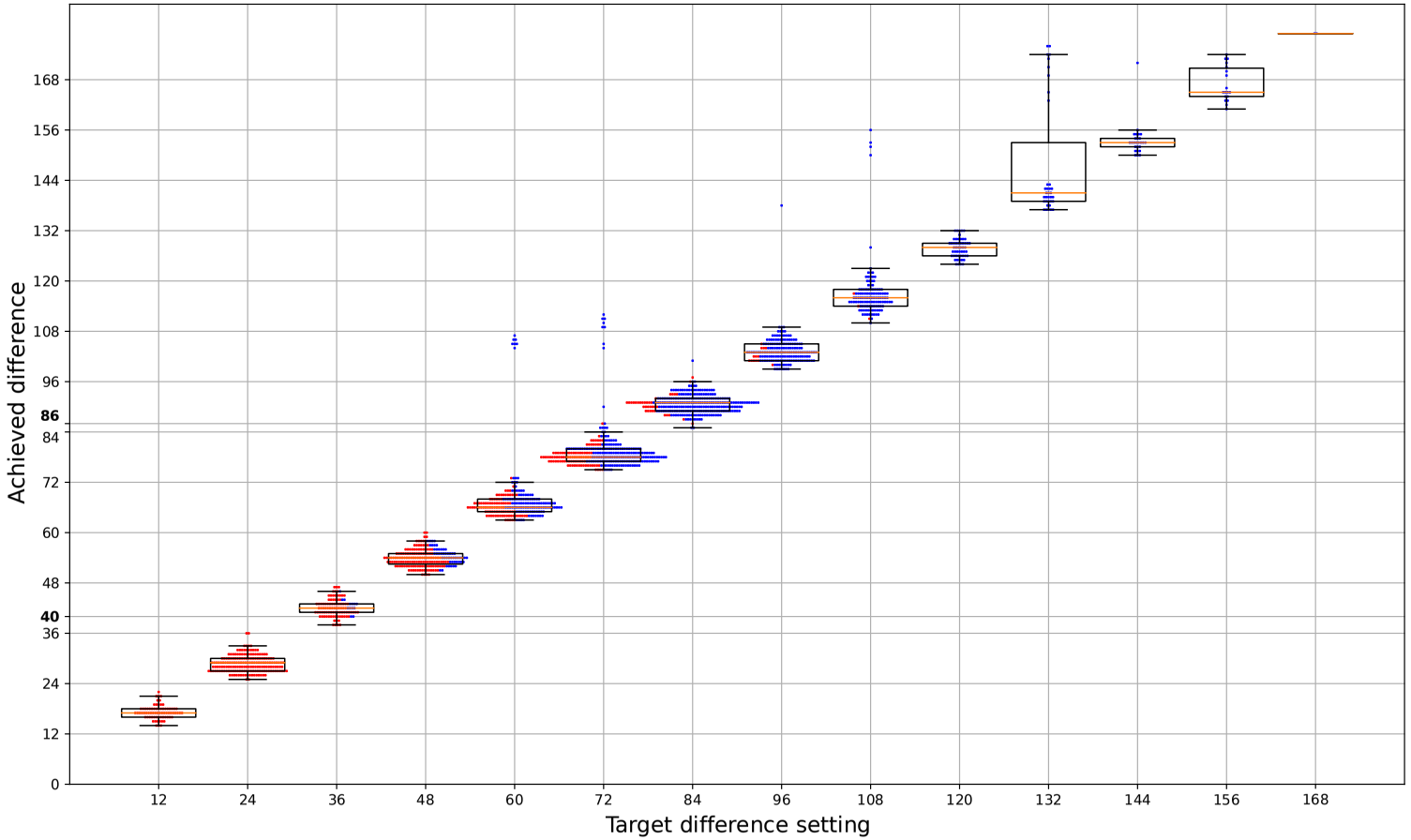

Figure 3.3: Final results of adaptively patching each of the 2000 ARM samples. Samples are only evaluated and displayed at the maximum setting they were successfully patched at. Samples are marked with different colors representing whether they where detected by SIMBIoTA or evaded detection (see Section 3.2.4).

if an executable can be modified both preserving its functionality and achieving a TLSH difference ≥ 86 from its original version (see section 5 of [14]).

The first setting of our attack that guarantees at least 86 points of self difference is setting 96; however, most binaries patched with setting 84 also exceed 86 in achieved self difference (see Figure 3.3). In total 710 of 2000 binaries (35.5%) were successfully patched in a way that their self difference was ≥ 86.

3.2.4 Adversarial examples to SIMBIoTA

SIMBIoTA [22] is a malware detection solution for IoT devices. Malware detection on IoT devices is problematic, because IoT devices have very limited resources regarding computational power, memory, and storage, so they do not meet the heavy requirements of traditional malware detection solutions. To fit these constraints, SIMBIoTA does similarity based malware detection on IoT devices, which entails the use of the TLSH similarity hash and difference score.

The computation done on the IoT device to classify a binary as malware or non-malware is as simple as calculating its TLSH hash, which is then compared to the TLSH hashes of a set of malware samples. These malware hashes are picked by and received from an antivirus server. A new binary is classified as malware if its TLSH difference from one of the TLSH hashes of malware is under a given threshold. The threshold used in SIMBIoTA is 40 points of difference score. Note that this is well beneath the proposed 86 in [14].

The server has a large database of malware samples and their TLSH hashes. Whenever the database is updated by adding new malware, which is inevitable to keep the pace with the constant evolution of malware, it computes a preferably small dominating set of all malware in the database. A dominating set is a subset of the greater set, which has at least one similar sample to any sample in the greater set. When the client IoT device requests an updated set of malware hashes, the server sends the hashes of its current dominating set. Hence, the device with a fresh set of hashes is guaranteed to recognize any malware, which is present in the server's database.

The relatively small size of the dominating set suggests that malware form crowded clusters of similar samples, and thus new samples can be detected with high probability. Indeed, SIMBIoTA achieved 90% detection rate on previously unseen malware, while its false positive rate stayed 0%. These rates were measured in a realistic simulation, where the client received weekly updates of the dominating hash set, and malware was "released" on the simulated timeline based on the time of its actual first submission to VirusTotal[4].

If we were to create adversarial examples to SIMBIoTA by patching malware samples in a way that the modified versions evade its detection, we would have to guarantee that its TLSH hash is not similar (diff > 40) to any of the hashes in the current dominating set. While this is quite a complex task even if the attacker knows the dominating set, evading detection can be made very probable by simply patching the sample with a high target self difference. This works because of the same reason SIMBIoTA is effective against unknown malware: malware tend to form dense clusters of similar samples [6]. If the modified version is very different from the original version, it is likely to be outside of its cluster, not being similar to any samples within, while it is also unlikely to become similar to a sample from another cluster.

[4] `https://virustotal.com` Last accessed: June 27, 2023

Consider a version of SIMBIoTA that has seen exactly the 29,209 ARM samples of CUBE-MALIoT-2021. Due to the dominating set mechanism, it would be guaranteed to detect any of the 29,209 samples; however, it would not detect samples that are similar to some of the 29,209, but not to any of the chosen dominating set. We test our patched samples by comparing them to each and every one of the 29,209 samples, and thus we classify all samples detected that might be detected with any dominating set selection. As we conducted our tests on a subset of the CUBE-MALIoT-2021 ARM samples, the original version of our patched samples are all present in the 29,209 known samples.

Let's evaluate our method as a tool to create adversarial examples to this simulated version of SIMBIoTA: From the 2000 input malware samples, 221 did not have section header tables, and thus the method had no chance to modify them. From the other 1779 samples, 1722 were successfully modified to some extent of self difference, including 1005 patched samples (50.3% of all, 56.5% of the samples that had a section header table) that evaded the detection of SIMBIoTA.

Figure 3.3 presents these detection results, marking detected and evading patched versions differently. As the original versions are all known to the simulated antivirus, no patches below the self difference of 40 points can evade its detection. Some patches not far past this threshold are already undetected, and with higher self differences the detection rates gradually drop to zero. By regression over this data, the attacker can approximate the probability of detection on any single patched sample.

3.2.5 Discussion

We presented and evaluated our attack against the fuzzy blacklisting of ELF binaries with default TLSH; however, the same approach should interfere with clustering and closest match search as well. While currently these seem to be the most common applications and configuration of TLSH, in this section we discuss our speculations on adaptation possibilities of the attack against others.

3.2.5.1 Other configurations of TLSH

There are a couple of variables in the configuration of TLSH; however, as stated in Section 2.1, practically only the defaults are used. In spite of this, it is interesting how these settings would affect our attack. The two officially supported adjustable settings in the build-time configuration[5] of the reference implementation are checksum size – 1 or 3 bytes, and number of buckets – 128 or 256, with the first option beeing default for both.

The checksum in TLSH is an attempt to identify completely indentical files; however it can be easily changed to any desired value by changing a single byte of the file. Using a bit longer checksum makes this a bit more difficult, but it should still be achievable by changing a few bytes only. As the difference of checksums can only contribute a single point to the TLSH difference score, this is irrelevant to our attack.

The number of buckets used is more interesting, as using 256 buckets (i.e. not discarding the upper 128 bucket counts) disables neutral patterns, as the addition of any sequence of n bytes will increase the bucket counts by $6n$ in total. While the use of neutral patterns might seem crucial in our attack, it is only a convenience. We use the neutral patterns as separator between the applied patterns and to fill up the rest of the modification window after the last pattern (see Section 3.1.3.4). While the separator bytes could be anything

[5] `https://github.com/trendmicro/tlsh#building-tlsh` Last accessed: June 23, 2023

without harmful bucket count contribution in themselves, without neutral patterns we could not trivially fill the rest of the modification window if we had found a shorter solution to achieve the desired bucket count contributions.

3.2.5.2 Other applications of TLSH

We showed that most ELF binaries can be patched in a way that the result has a high TLSH difference from the original version even without changing the size of the file. The requirement that enabled this attack was the fact that the binary had continuous regions that the attacker could overwrite without affecting the program's behavior. Patching any other type of file with this same method has the same requirement: containing one or more continuous ranges of bytes that the attacker can freely modify without damaging the essential semantic content of the file. As a few examples how this might be possible, file formats may enable adding bytes at the end of the file which are later ignored, adding metadata that is never validated, or adding comments. Also note that the pattern construction might be successful even if the allowed bytes or byte sequences are limited (e.g. printable ASCII only).

Security of some other proposed applications of TLSH depend on the robustness of the scheme just as it is the case with malware detection. One of these is the system described in [8], a blockchain based decentralized searching solution, where data owners receive payment proportionally to the number of provided search results. The authors adopt TLSH to ensure that greedy providers do not duplicate matching results to receive double payment. Another one is [1], where accesses to confidential documents are logged along with the TLSH hashes of the accessed documents, for in case of document leakage the perpetrator can be more easily identified through comparing the hash of the leaked document to the ones in the access log and identifying the entries that are most likely the source of the leak even if the corresponding document was somewhat edited between access and the leak. Neither of these proposals detail the type of files that are compared using TLSH, both referring to them as "documents". So they might be vulnerable depending on whether the file type enables the above discussed way of editing without consequences.

Another class of applications is fuzzy whitelisting. Similarity digest schemes can be used in whitelisting to allow not only exact matches, but similar files to the file hashes of the whitelist as well. Whitelisting software binaries with TLSH is discussed in [17]. A more complex whitelisting application is [4], where the network traffic of IoT devices is inspected to detect anomalous (probably malicious) activity. This is possible because such devices have limited functionality, thus their network traffic flow follows usual patterns. Similarity digest schemes are utilized to compare current traffic to recorded benign traffic patterns. While our anti-blacklisting attack required modifying malicious files in any way that their new version had a high TLSH difference from the original, an anti-whitelisting attack would require modifying the files so they become similar (i.e. have low difference score) to the allowed file hashes of the whitelist. This is exactly what our second attack discussed in Chapter 4 aims to achieve.

Chapter 4

Attack II: anti-whitelisting

Making the unsimilar look similar.

In this chapter we present and evaluate our second attack, which aims to modify binary files in a way that their TLSH hash will be very similar to another arbitrary TLSH hash. While in the previous attack presented in Chapter 3 trying to maximize TLSH difference we did not have an easily definable optimal outcome as "perfect unsimilarity" does not make sense, this time we have: we can define "perfect similarity" as a perfect TLSH hash collision.

While this attack usually requires a far greater extent of modification to the files, it is a much stronger attack as well. In the case of the previous "anti-blacklisting" attack, if the defender knows our method of modification, they can for example proactively blacklist hashes of modified samples as well to mitigate the attack. By this "anti-whitelisting" attack we can take any two samples that should result in a different decision on the defender's end (e.g. a malware sample and a benign executable) and modify the one that should be blacklisted (e.g. detected as malware) to make its TLSH hash be the exact same as that of the other. As an evaluation to the attack, we show a practical example to this in Section 4.2. With such possible hash collisions it is impossible to reason for a decision based on the TLSH hashes of the files alone. For example, if a malware sample has the exact TLSH hash of a well-known benign executable, it is impossible to detect it as malware based on its TLSH hash alone without making the same decision for the benign file. For this reason this attack is also guaranteed to be effective against alternative applications of the TLSH hashes like SIMBIoTA-ML [15], which make decisions based on TLSH hashes without any use of the TLSH difference score calculation.

4.1 Design

We managed to succeed with the first attack in most cases by only overwriting unused parts of the binaries and thus not changing the file size at all. This case the required changes are of a much bigger extent, thus we fall back to the simple strategy of appending to the end of the files. It is arguable that such additions to files can be quite easily detected, however an attacker might be able to hide the added content in many creative ways.

4.1.1 Controlling the hash fields

To modify a file in a way that its TLSH hash will completely match another given hash (*target hash*) we need to be able to precisely control the value of each and every hash field. The hash fields are the `checksum`, the `lvalue`, the `Q1ratio` and `Q2ratio` and the `codes`. The `lvalue` and `checksum` fields can be quite easily manipulated (see sections 4.1.4 and 4.1.5), the more interesting ones are the `Q1ratio`, `Q2ratio` and `codes` as they are controlled by the relations of the bucket counts.

Our strategy is to calculate target ranges for each bucket count so if all their values fall into the corresponding target range it guarantees the same values for `Q1ratio`, `Q2ratio`, and `codes` as those of the target hash. As we can only increase bucket counts from their current values by addition to the file each of these target ranges should enable higher values then the current value of the corresponding bucket count.

4.1.2 Calculating the bucket count constraints

From the `codes` of the target hash we can see exactly which buckets should belong to which quartile intervals. From this information we can set a few lower constraints for our target quartile values. As we can only increase the current bucket count values by appending, if a bucket count is currently b_i and $\texttt{code}_i = 0$ in the target hash (i.e. $b_i \leq q_1$), we know that our new q_1 has to be $\geq b_i$. Let's call the maximum of these lower bounds $q_{1,\text{start}}$, and define $q_{2,\text{start}}$ and $q_{3,\text{start}}$ similarly.

$$q_{1,\text{start}} := \max\{b_i : \texttt{code}_{i,\text{target}} = 0\} \tag{4.1}$$

$$q_{2,\text{start}} := \max\{b_i : \texttt{code}_{i,\text{target}} = 1\} \tag{4.2}$$

$$q_{3,\text{start}} := \max\{b_i : \texttt{code}_{i,\text{target}} = 2\} \tag{4.3}$$

Note that it is not guaranteed that $q_{1,\text{start}} \leq q_{2,\text{start}} \leq q_{3,\text{start}}$.

Next, let's see what is required to achieve the target quartile ratios. We can retrieve `Q1ratio` and `Q2ratio` from the target hash, which are mod16 stored values of $\lfloor \frac{q_1}{q_3} \cdot 100 \rfloor$ and $\lfloor \frac{q_2}{q_3} \cdot 100 \rfloor$ respectively. Functions dqr_1 and dqr_2 map the ratio of planned quartiles to the difference of the corresponding target value and their resulting encoded quartile ratio value based on their current values.

$$\text{dqr}_1 : \frac{q_1}{q_3} \mapsto (\texttt{Q1ratio}_{\text{target}} - \lfloor \frac{q_1}{q_3} \cdot 100 \rfloor) \bmod 16 - 8 \tag{4.4}$$

$$\text{dqr}_2 : \frac{q_2}{q_3} \mapsto (\texttt{Q2ratio}_{\text{target}} - \lfloor \frac{q_2}{q_3} \cdot 100 \rfloor) \bmod 16 - 8 \tag{4.5}$$

Algorithm 1 calculates our final target quartile values with the required ratios. The outer loop is required because with smaller values sometimes increasing one of the quartile values by one to change the ratio in a given direction can already cause the ratio to change by more than one percent sometimes resulting in skipping the ratio value that would result in the desired `Q1ratio` or `Q2ratio` value. Because each modification increments one of the quartile values, it will cease this issue sooner or later.

It is still not guaranteed that $q_1 \leq q_2 \leq q_3$; however, in all practical cases we encountered so far in our experiments it was the case, thus even though it might be easy to solve this potential issue by adjusting the algorithm, our implementation of the attack simply contains an assertion for this to be true.

Algorithm 1 Achieving the target quartile ratios with planned quartile values

$q_1 \leftarrow q_{1,\text{start}}$
$q_2 \leftarrow q_{2,\text{start}}$
$q_3 \leftarrow q_{3,\text{start}}$
while $\text{dqr}_1(\frac{q_1}{q_3}) \neq 0$ or $\text{dqr}_2(\frac{q_2}{q_3}) \neq 0$ **do**
    while $\text{dqr}_1(\frac{q_1}{q_3}) < 0$ or $\text{dqr}_2(\frac{q_2}{q_3}) < 0$ **do**
        $q_3 \leftarrow q_3 + 1$
    while $\text{dqr}_1(\frac{q_1}{q_3}) > 0$ **do**
        $q_1 \leftarrow q_1 + 1$
    while $\text{dqr}_2(\frac{q_2}{q_3}) > 0$ **do**
        $q_2 \leftarrow q_2 + 1$
return q_1, q_2, q_3

Having calculated the appropriate quartile q_1, q_2, q_3 values we can set the target ranges for each of the bucket counts.

$$\texttt{bc\_bound}_i = \begin{cases} [0, q_1] & \text{if } \texttt{code}_{i,\text{target}} = 0 \\ (q_1, q_2] & \text{if } \texttt{code}_{i,\text{target}} = 1 \\ (q_2, q_3] & \text{if } \texttt{code}_{i,\text{target}} = 2 \\ (q_3, \text{inf}) & \text{if } \texttt{code}_{i,\text{target}} = 3 \end{cases} \tag{4.6}$$

One more requirement is to have some bucket counts for each quartile that actually hold the exact value of the given quartile. This can be achieved by tightening the $\texttt{bc\_bound}_i$ range corresponding to the given bucket to $[q_x, q_x]$. Usually this affects one bucket count for each quartile; however, there are some rare exceptions. These rare cases are when many bucket counts share the same value around the transition between quartiles. For example, if in the ascending (i.e. sorted) order of the bucket counts the 32nd count equals the 33rd and 34th, they will all map to 0 in `codes`, because even though the 33rd and 34th are past the 32nd count, which defines q_1, their values are still $\leq q_1$. If we have this phenomenon in the target hash, we have to replicate it by constraining multiple bucket counts to hold the exact value of the given quartile. We choose to always tighten the bounds for the already highest bucket count (or counts) belonging to each of the first three quartile intervals.

4.1.3 Achieving the required bucket counts

Having determined the target ranges ($\texttt{bc\_bound}_i$) for each bucket count we want to achieve, we could use a similar approach with periodic patterns as described in Sections 3.1.2 and 3.1.3. Instead, we used a less sophisticated, but very fast greedy generation technique.

The idea is that we try each possible byte value to append to the file data and track the bucket count contributions of the (six) byte triads newly formed with the last (four) previous bytes. Let's call these the contribution of the new byte (value). Each byte value is disqualified if it makes a contribution that raises a bucket count above its desired maximal value, otherwise it is scored to its total contribution to the bucket counts that are still below their desired minimal value. We greedily take the highest scoring byte value, append it to the file data, and proceed to generate the next one until we either succeed to land every bucket count in the target ranges, or fail to find any byte value at some point that is not disqualified by its undesired contributions.

4.1.4 Achieving the required file size

By achieving bucket counts from the target ranges, we guaranteed the required values for the quartile ratios and the codes. `lvalue` depends only on the file size. While we cannot make the file smaller by appending, we can easily make it bigger. The only challenge is to do so without changing the previously set bucket counts. The trick is simple: we append a neutral periodic pattern (see Section 3.1.2.2) until we reach the required `lvalue`. While the inner byte triads of the neutral pattern are guaranteed not to contribute to any bucket counts used for the hash computation, placing it into context the byte triads formed from bytes of both the context and the pattern have no such guarantees. To overcome this challenge we do a breath first search (BFS) of bytes to append before the neutral periodic pattern to find the shortest "bridge" of bytes so the contributions do not increase any bucket counts above their desired maximal values. Once we found such a bridge, we can append any number of iterations of the neutral pattern, and thus achieve the required file size as well.

4.1.5 Faking the checksum

If we followed through and succeeded with all the previous steps above, we are guaranteed to have an almost perfect hash collision. The only hash field that is not guaranteed to match the corresponding value of the target hash is the `checksum`. This means that the TLSH difference between the hash of our modified file and the target hash is already only either 0 or 1.

The value of the one-byte checksum can be changed to any value by appending just a single byte (see Figure 2.1). As this might change some already fine tuned bucket counts in an undesired way, we use a very similar BFS to the one we used for finding the bridge in Section 4.1.4 instead, to find the shortest sequence of bytes to append that changes the checksum to its target value while keeping all the bucket counts within their target ranges at the same time.

At this point we should have a complete TLSH hash collision.

4.2 Evaluation

4.2.1 Making any malware look like GCC

For evaluation of the anti-whitelisting attack we used the same 2000 IoT malware samples from the CUBE-MALIoT-2021 dataset as in the evaluation of the anti-blacklisting attack (Section 3.2). Due to the limitation of only being able to increase file sizes, and thus being able to mutate files to perfectly match the hashes (more specifically the `lvalue`) of larger files only (see Section 4.1.4), we chose a common target hash of a file larger than almost all malware samples. This chosen target was the hash of a version of the GCC compiler executable for the same 32-bit ARM architecture, which is just over a mebibyte in size, larger than 1991 of the 2000 randomly selected malware samples. It is a very well known benign executable, which under no circumstances should ever be classified as malware by any antivirus; therefore, if we can modify malware samples to have the exact same TLSH it has, classifying them will be impossible based on their TLSH hashes alone.

To have more meaningful results we split the attack into two phases: faking the bucket count related values (`codes`, `Q1ratio`, and `Q2ratio`); and faking the `lvalue` and `checksum`.

This way we can capture how much addition is required to fake everything but basically the `lvalue`, as faking the checksum takes appending only a few bytes. While many times we might not even care about the `lvalue`, this is meaningful even if we always want perfect collisions, because this shows how much bigger the target file should be for probable success.

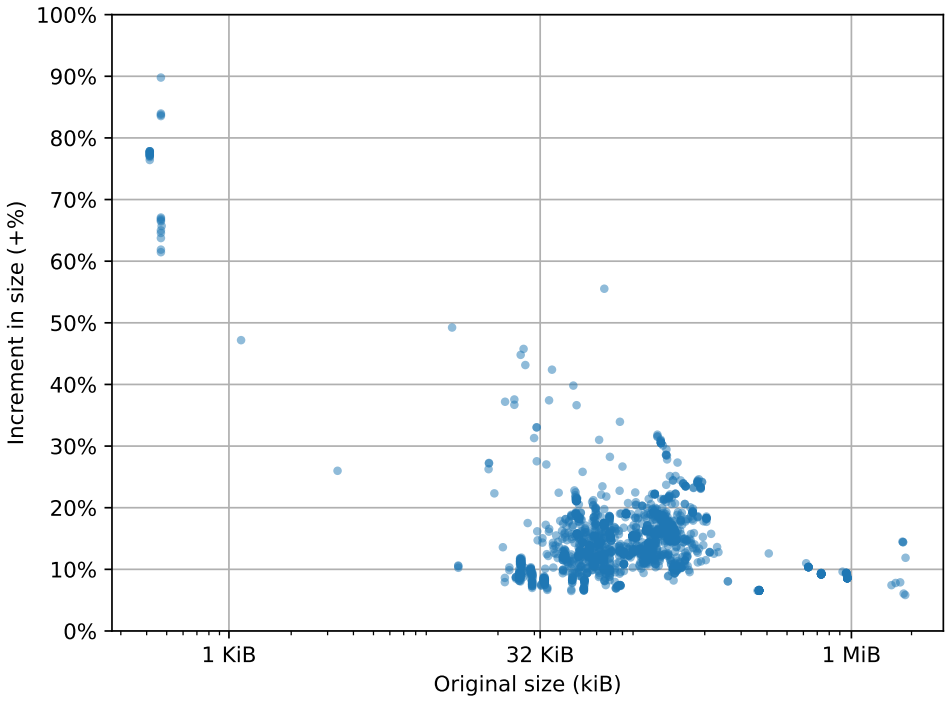

Figure 4.1: Increments in sizes of the 2000 malware samples by appending the specially crafted data to achieve collisions of bucket count related TLSH hash fields with the corresponding values in the hash of the GCC executable.

Figure 4.1 showcases these first phase increments for the 2000 malware samples. Other than in the cases of really small samples and a few other exceptions, the required increment tends to be around 10-20% ($< 20\%$ in 90% of cases, $< 23\%$ in 95% of cases). This is about 10 times more than what we usually modified in the first attack (i.e. the usual size of section header tables in the ELF binaries).

After executing the second phase of the attack as well for each sample, all 1991 samples (99.55%) that were originally smaller than GCC (none of which became larger than GCC in the first phase either) resulted in having the exact same 35-byte TLSH hash of GCC, while the 9 samples that were already larger (by at least +70%) resulted in having the exact same values for every hash field other than the `lvalue`.

4.2.2 Discussion

4.2.2.1 Other configurations of TLSH

As discussed in Section 3.2.5.1, the two main alterable parameters of the TLSH algorithm are the effective bucket count (128 or 256) and the checksum size (1 or 3 bytes). Here we speculate how these possible changes would affect the attack.

Using all 256 buckets would change the attack quite a bit, as we could not separate the two phases of achieving the desired bucket count relations and achieving the desired file size, because in case of keeping all 256 bucket counts, there would be no neutral patterns; in fact, each additional byte appended to the file would increase the bucket counts by a constant 6 in total. Therefore, the target ranges for the bucket counts would already have to take the file size into account. Another way using all 256 bucket counts could make the attack more difficult is that currently the greedy generation might use neutral bytes instead of failing in some cases, which would become impossible in the same way.

A three byte checksum might be a bit more difficult to fake, possibly taking a few more steps of depth in the BFS.

4.2.2.2 Periodic patterns or greedy generation

We used two different techniques to manipulate the bucket counts in the two attacks. Theoretically both could be used in both scenarios. While this comparison is mostly future work, based on some early experiments we expect periodic patterns to be more effective in cases when a few selected bytes have to be increased by larger extents, while greedy generation should take on in more general cases. In our current implementations greedy generation is much faster and more easily configurable. Another speculation is that the greedily generated mostly random like byte patterns look less suspicious and are harder to be detected as intentional tempering than the more easily recognizable periodic patterns.

Greedy generation might be improved by brute forcing groups of bytes instead of single bytes, selecting the most promising and either jumping to the next entire block, or keeping just the first (few) bytes of the chosen block and generating the next one after. Another improvement could be, in case of more difficult tasks resulting in more failures, to implement backtracking in case of failures and choosing alternative locally optimal or even suboptimal solutions to mitigate the given failure.

Chapter 5

Conclusion

In this paper, we proposed two targeted attacks against the TLSH similarity digest scheme and conducted experiments regarding its robustness against said attacks. For evaluation, we used IoT malware samples, since malware detection and malware clustering are prominent use-cases of TLSH.

The first proposed attack modifies binaries by overwriting an unused portion of them (usually around 1-2% of the file in size), thus preserving their functionality, while creating the largest possible TLSH difference compared to the original files. Out of the 2000 malware samples that we tried to patch with this attack method, 1779 samples were suitable for modification; we were able to achieve the self difference score of at least 86 (a limit determined by the authors of TLSH) for 710 samples; and we could patch 1465 samples to have a self difference score of at least 40, which is the similarity threshold used by SIMBIoTA, a malware detection solution built on TLSH. Furthermore, in 1005 cases, the patched samples achieved a difference score of 40 to every other sample known by SIMBIoTA, thus they surely evade detection by this method. In summary, we were able to patch 40% of all modifiable samples to the point where even the authors of TLSH would consider the scheme broken, while in 56% of the cases, we could fool a malware detector built upon TLSH. In addition, our method does not change the size of the samples, it preserves their functionality, and it alters only one of the small modifiable portions of them.

The second attack modifies the TLSH hash of the file to a given arbitrary hash by appending specially crafted data to the end of the file. With this method, we managed to successfully modify 1991 of the same original 2000 malware samples to have the exact same hash as a well known benign executable, the main binary of GCC, which is present on a significant portion of Linux-based computers. This makes the detection of these modified samples as malware completely impossible based on their hashes alone. In the other 9 cases where our method failed to produce a complete hash collision, the size of the samples were too close to the size of the target benign executable, hence we did not have sufficient amount of space for our attack, and we could not achieve a match in the file size related field of the TLSH values. However, even in these cases, we succeeded to match values of all the other 131 hash fields. Thus, a practical requirement for achieving complete hash collision seems to be that the size of the target file must be somewhat larger than the size of the files we are trying to manipulate. The usual size of data appended to match the values of all other hash fields was $< 20\%$ of the original file size in 90% of all 2000 cases. In summary, our method achieved complete hash collisions with a well known benign file in more than 99.5% of the cases, and almost complete hash collision in the rest

of the cases as well. This means that no decision based on TLSH hashes alone can be considered trustworthy if an attacker can append specially crafted data to the end of files in an unnoticeable way.

Overall, we can conclude that, although TLSH is considered to be robust against random attacks, it is not robust at all against targeted attacks.

Finally, we note that TLSH is used by VirusTotal, the largest malware repository in the world, for similarity-based searches. Therefore, we notified VirusTotal about our attacks and we are awaiting their reaction. It may well be possible that TLSH needs to be abandoned and replaced by another, more robust similarity digest scheme. However, to the best of our knowledge, such a robust similarity digest scheme does not exist yet. Hence, in our future research, we set the ambitious goal of designing the first such scheme.

Acknowledgements

The research presented in this paper was supported by the European Union project RRF-2.3.1-21-2022-00004 within the framework of the Artificial Intelligence National Laboratory. The presented work also builds on results of the SETIT Project (2018-1.2.1-NKP-2018-00004), which was implemented with the support provided from the National Research, Development and Innovation Fund of Hungary, financed under the 2018-1.2.1-NKP funding scheme.

Bibliography

[1] Abdulrahman Alruban, Nathan Clarke, Fudong Li, and Steven Furnell. Biometrically linking document leakage to the individuals responsible. In *Trust, Privacy and Security in Digital Business: 15th International Conference, TrustBus 2018, Regensburg, Germany, September 5–6, 2018, Proceedings 15*, pages 135–149. Springer, 2018.

[2] F. Breitinger, H. Baier, and J. Beckingham. Security and implementation analysis of the similarity digest sdhash. In *Proceedings of the 1st International Baltic Conference on Network Security and Forensics (NeSeFo)*, 2012.

[3] Donghoon Chang, Mohona Ghosh, Somitra Sanadhya, Monika Singh, and Douglas White. Fbhash: A new similarity hashing scheme for digital forensics. *Digital Investigation*, 29:S113–S123, 07 2019. DOI: 10.1016/j.diin.2019.04.006.

[4] Batyr Charyyev and Mehmet Hadi Gunes. Detecting anomalous iot traffic flow with locality sensitive hashes. In *GLOBECOM 2020-2020 IEEE Global Communications Conference*, pages 1–6. IEEE, 2020.

[5] Jianxing Chen, Romain Fontugne, Akira Kato, and Kensuke Fukuda. Clustering spam campaigns with fuzzy hashing. In *Proceedings of the Asian Internet Engineering Conference (AINTEC)*, pages 66–73, 11 2014. DOI: 10.1145/2684793.2684803.

[6] E. Cozzi, P.A. Vervier, M. Dell'Amico, Y. Shen, L. Bilge, and D. Balzarotti. The tangled genealogy of IoT malware. In *Proceedings of the Annual Computer Security Applications Conference*, pages 1–16, 2020. DOI: 10.1145/3427228.3427256.

[7] Ernesto Damiani, Sabrina De Capitani di Vimercati, Stefano Paraboschi, and Pierangela Samarati. An open digest-based technique for spam detection. In *PDCS*, pages 559–564. Citeseer, 2004.

[8] Meiqi He, Gongxian Zeng, Jun Zhang, Linru Zhang, Yuechen Chen, and SiuMing Yiu. A new privacy-preserving searching model on blockchain. In *Information Security and Cryptology–ICISC 2018: 21st International Conference, Seoul, South Korea, November 28–30, 2018, Revised Selected Papers 21*, pages 248–266. Springer, 2019.

[9] J. Kornblum. Identifying almost identical files using context triggered piecewise hashing. In *Proceedings of the 6th Annual DFRWS Confenerce*, 2006.

[10] Yuping Li, Sathya Chandran Sundaramurthy, Alexandru G. Bardas, Xinming Ou, Doina Caragea, Xin Hu, and Jiyong Jang. Experimental study of fuzzy hashing in malware clustering analysis. In *Proceedings of the 8th USENIX Conference on Cyber Security Experimentation and Test (CSET)*. USENIX Association, 2015.

[11] Miguel Martín-Pérez, Ricardo J Rodríguez, and Frank Breitinger. Bringing order to approximate matching: Classification and attacks on similarity digest algorithms. *Forensic Science International: Digital Investigation*, 36:301120, 2021.

[12] Jonathan Oliver and Josiah Hagen. Designing the elements of a fuzzy hashing scheme. In *2021 IEEE 19th International Conference on Embedded and Ubiquitous Computing (EUC)*, pages 1–6. IEEE, 2021.

[13] Jonathan Oliver, Chun Cheng, and Yanggui Chen. TLSH – A Locality Sensitive Hash. In *2013 Fourth Cybercrime and Trustworthy Computing Workshop*, pages 7–13, Sydney NSW, Australia, November 2013. IEEE. ISBN 978-1-4799-3076-0 978-1-4799-3075-3. DOI: 10.1109/CTC.2013.9. URL http://ieeexplore.ieee.org/document/6754635/.

[14] Jonathan Oliver, Scott Forman, and Chun Cheng. Using randomization to attack similarity digests. In *International Conference on Applications and Techniques in Information Security*, pages 199–210. Springer, 2014.

[15] Dorottya Papp, Gergely Ács, Roland Nagy, and Levente Buttyán. SIMBIoTA-ML: Light-weight, machine learning-based malware detection for embedded IoT devices. In *Proceedings of the 7th International Conference on Internet of Things, Big Data and Security (IoTBDS)*, pages 55–66. INSTICC, SciTePress, 2022. ISBN 9789897585647. DOI: 10.5220/0000159700003194.

[16] Peter K Pearson. Fast hashing of variable-length text strings. *Communications of the ACM*, 33(6):677–680, 1990.

[17] Jayson Pryde, Nestle Angeles, and Sheryl Kareen Carinan. Dynamic whitelisting using locality sensitive hashing. In *Trends and Applications in Knowledge Discovery and Data Mining: PAKDD 2018 Workshops, BDASC, BDM, ML4Cyber, PAISI, DaMEMO, Melbourne, VIC, Australia, June 3, 2018, Revised Selected Papers 22*, pages 181–185. Springer, 2018.

[18] V. Roussev. Data fingerprinting with similarity digests. In *Research Advances in Digital Forensics VI*, pages 207–226, 2010.

[19] V. Roussev. An evaluation of forensics similarity hashes. In *Proceedings of the 11th Annual DFRWS Confenerce*, 2011.

[20] Nikolaos Sarantinos, Chafika Benzaid, Omar Arabiat, and A. Al-Nemrat. Forensic malware analysis: The value of fuzzy hashing algorithms in identifying similarities. pages 1782–1787, 08 2016. DOI: 10.1109/TrustCom.2016.0274.

[21] József Sándor, Roland Nagy, and Levente Buttyán. Increasing the robustness of a machine learning-based iot malware detection method with adversarial training. In *Proceedings of the 2023 ACM Workshop on Wireless Security and Machine Learning (WiseML '23)*, 2023.

[22] Csongor Tamás, Dorottya Papp, and Levente Buttyán. SIMBIoTA: Similarity-based malware detection on IoT devices. In *Proceedings of the 6th International Conference on Internet of Things, Big Data and Security (IoTBDS)*, pages 58–69. INSTICC, SciTePress, 2021. ISBN 978-989-758-504-3. DOI: 10.5220/0010441500580069.