A Practical Attack on the TLSH Similarity Digest Scheme

Gábor Fuchs
gfuchs@crysys.hu
Budapest University of Technology and Economics
Budapest, Hungary

Roland Nagy
rnagy@crysys.hu
Budapest University of Technology and Economics
Budapest, Hungary

Levente Buttyán
buttyan@crysys.hu
Budapest University of Technology and Economics
Budapest, Hungary

ABSTRACT

Similarity digest schemes are used in various applications (e.g., digital forensics, spam filtering, malware clustering, and malware detection), which require them to be resistant to attacks aiming at generating semantically similar inputs that have very different similarity digest values. In this paper, we show that TLSH, a widely used similarity digest function, is not sufficiently robust against such attacks. More specifically, we propose an automated method for modifying executable files (binaries), such that the modified binary has the exact same functionality as the original one, it also remains syntactically similar to the original one, yet, the TLSH difference score between the original and the modified binaries becomes high. We evaluate our method on a large data set containing malware binaries, and we also show that it can be used effectively to generate adversarial samples that evade detection by SIMBIoTA, a recently proposed similarity-based malware detection approach.

KEYWORDS

Similarity digest schemes, locality sensitive hashing, TLSH, similarity-based malware detection

ACM Reference Format:
Gábor Fuchs, Roland Nagy, and Levente Buttyán. 2023. A Practical Attack on the TLSH Similarity Digest Scheme. In *The 18th International Conference on Availability, Reliability and Security (ARES 2023), August 29–September 01, 2023, Benevento, Italy*. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3600160.3600173

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 [17], Nilsimsa [7], and TLSH [14] are used in, for instance, digital forensics [3], spam filtering [5], malware clustering [10, 19], and malware detection [21]. In each of these applications, similarity of various files are measured based on the similarity of their digest values.

ARES 2023, August 29–September 01, 2023, Benevento, Italy

ACM ISBN 979-8-4007-0772-8/23/08…$15.00
https://doi.org/10.1145/3600160.3600173

In many applications, including the ones mentioned above, 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 spam filtering and malware detection; and similarly, file similarity analysis in forensic investigations and clustering malware samples based on their similarity digests would make no sense. For this reason, the robustness of existing similarity digest schemes against such attacks have been extensively studied [2, 11, 13, 18], and the authors of [13] 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 an automated method for modifying 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. Our method does not involve any encryption or packing techniques: it preserves 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, yet, the difference score returned by the official TLSH difference calculation function for the digests of the modified and the original binaries is high. Our method heavily relies on the details of how TLSH computes similarity hashes and how the TLSH difference score is calculated. As a practical application of our 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 [21]. Finally, we note that our method 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 section 2, we introduce the necessary background on the operation of the TLSH similarity digest scheme, we discuss its robustness analysis presented in [13], and we introduce some details of the ELF file format. In section 3, we present the design of our 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 4, 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. Finally, in section 5, we conclude our paper.

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 attack against TLSH is similar in nature, it requires detailed discussion of the TLSH algorithm.

2.1 The TLSH algorithm

In this section we discuss the TLSH algorithm. Note that its presentation in [12] 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 [14], 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, 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 1).

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

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

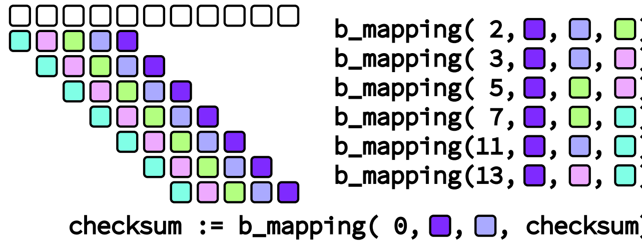

Figure 1: The TLSH sliding window and the calculations with the selected bytes: the formula for the 6 single-byte hash values, and the update of the checksum value for each window position. `b_mapping` is a deterministic hash function based on the Pearson hash [15].

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

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`: $\texttt{diff}_{\texttt{checksum}} = \begin{cases} 0 & \text{if } \texttt{checksum}_a = \texttt{checksum}_b \\ 1 & \text{if } \texttt{checksum}_a \ne \texttt{checksum}_b \end{cases}$
- `lvalue`: having $\texttt{ldiff} = \texttt{mod\_diff}(\texttt{lvalue}_a, \texttt{lvalue}_b, 256)$; (the use of `mod_diff` is interesting here[4])

[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.
[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`

$$\texttt{diff}_{\texttt{lvalue}} = \begin{cases} \texttt{ldiff} & \text{if } \texttt{ldiff} \leq 1 \\ 12 \cdot \texttt{ldiff} & \text{if } \texttt{ldiff} > 1 \end{cases}$$

- 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 [13]. 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 [13], 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 [13]).

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

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. 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.

3 DESIGN

Our ultimate goal is to modify ELF binaries in such a way that

(1) their functionality remains unaffected;
(2) their size remains unchanged; and

0 and 169 (ldiff = 87) being scored much less different than lvalue 0 and 128 (ldiff = 128).

[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

(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 [20]). 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 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.

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 $2 \le d_1, d_2 \le 8$ and $(d_1 - 1) + (d_2 - 1) \ge \lceil \frac{t}{12} \rceil$, where t is the target self difference score to achieve. This would result in at least $12 \cdot \lceil \frac{t}{12} \rceil \ge t$ 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 $32 < t \le 48$, 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 2). 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.

3.2 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.

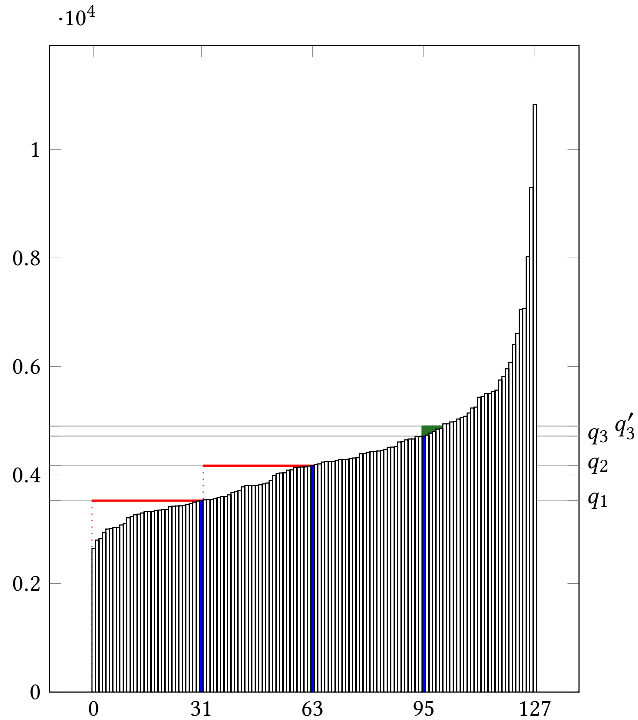

Figure 2: The sorted bucket counts for one of the tested malware samples. Highlighted are the quartile defining counts, the described minimal required changes to increase q_3 to the target value, and the bucket count limits that must not be exceeded not to increase q_1 or q_2.

3.2.1 *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 2, 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: $+n$.

3.2.2 *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.

3.2.3 *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 3.3.4. 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.

3.3 The steps of patching

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

3.3.1 *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[7] from the data before any modifications. Let's call these $r_{1,\text{orig}}$ and $r_{2,\text{orig}}$.

3.3.2 *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.

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.

3.3.3 *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. 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 and Inequality 6. This can be done by trying each value increasingly, beginning from the current q_3 value.

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

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

$$2 \leq d_i \leq 8 \qquad i = 1, 2 \tag{3}$$

$$\texttt{diff}_{\texttt{Q1ratio}} = (d_1 - 1) \cdot 12 \tag{4}$$

$$\texttt{diff}_{\texttt{Q2ratio}} = (d_2 - 1) \cdot 12 \tag{5}$$

$$\texttt{diff}_{\texttt{Q1ratio}} + \texttt{diff}_{\texttt{Q2ratio}} \geq t \tag{6}$$

Now that we have q_3', we calculate the required bucket count increments exactly as explained in Section 3.1 and visualized in Figure 2.

3.3.4 *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

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

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.

3.3.5 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 (7) and (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 ∞.

$$t_i \leq \sum_j \left(n_j c^P_{j,i} + \begin{cases} 0 & \text{if } n_j = 0 \\ c^S_{j_i} & \text{if } n_j > 0 \end{cases} \right) \leq l_i \qquad (7)$$

$$4 + \sum_j \left(n_j p_j + \begin{cases} 0 & \text{if } n_j = 0 \\ 4 + 4 & \text{if } n_j > 0 \end{cases} \right) \leq L \qquad (8)$$

Inequality (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 (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 (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.

3.3.6 Choose and fill the patterns – Solution. To solve the system of inequalities (7) and (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[8] 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 2).

So we find the few patterns with the highest scores, and we use a solver to solve the system of inequalities (7) and (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 3.3.4.

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.

4 EVALUATION

4.1 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[9]. 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).

[8] 32 or 16 in our experiments
[9] https://github.com/CrySyS/CUBE-MALIoT-2021 Last accessed: June 27, 2023

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

In Figure 3, 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 4.

4.3 Comparison to other research

As reviewed in Section 2.2, the robustness of TLSH (along with other similarity hashes) was tested in [13] on a few kinds of data including executable files.

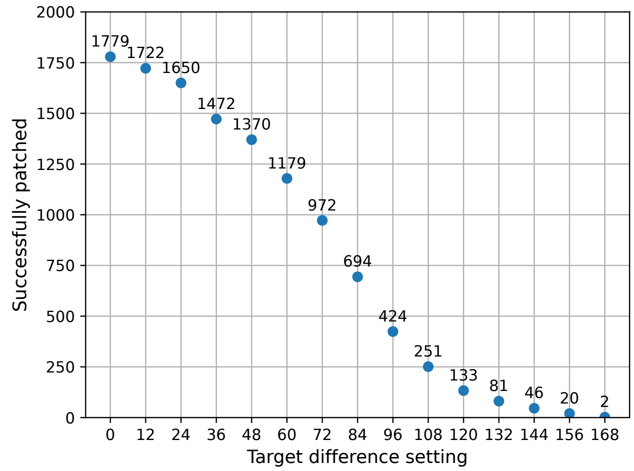

Figure 3: Results on 2000 ARM samples. Each value represents how many samples were successfully patched on the given or a higher setting. Setting 0 refers to the number of all samples that had a section header table to work with.

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 if an executable can be modified both preserving its functionality and achieving a TLSH difference ≥ 86 from its original version (see section 5 of [13]).

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 4). In total 710 of 2000 binaries (35.5%) were successfully patched in a way that their self difference was ≥ 86.

4.4 Adversarial examples to SIMBIoTA

SIMBIoTA [21] 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

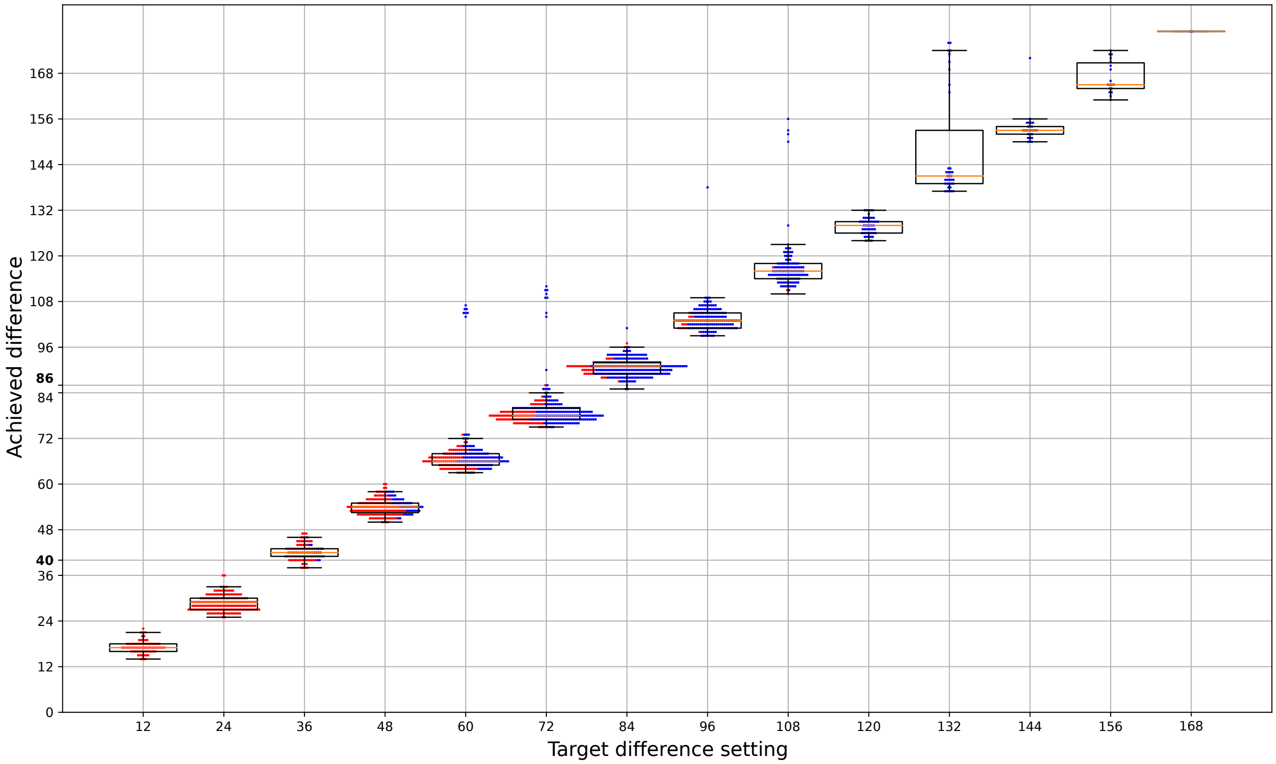

Figure 4: Final results of adaptively patching each of the 2000 ARM samples. Samples are only evaluated and displayed at the maximum setting they were successfully patched at. Samples are marked with different colors representing whether they where detected by SIMBIoTA or evaded detection (see Section 4.4).

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 [13].

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[10].

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.

[10]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 4 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.

4.5 Discussion – Other applications and configuration of TLSH

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.

4.5.1 Other configurations. 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[11] 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 3.3.4). While the separator bytes could be anything 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.

4.5.2 Other applications. 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 refering 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 [16]. 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 something we are planning to work on in the near future.

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

5 CONCLUSION

In this paper, we propose a targeted attack against the TLSH similarity digest scheme and conduct experiments regarding its robustness against said attack. For evaluation, we used IoT malware samples, since malware detection and malware clustering are a prominent use-cases of TLSH. The proposed attack attempts to modify binaries by overwriting an unused portion of the binary, thus preserving its functionality, while creating the largest possible TLSH difference compared to the original file. Out of the 2000 malware samples that we tried to patch with our attack method, 1779 samples were suitable for modification; we were able to achieve the self difference 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 a similarity threshold used by SIMBIoTA, a malware detection solution built on TLSH. Furthermore, in 1005 cases the patched samples achieved a 40 difference score to every other sample known by SIMBIoTA, thus they surely evade the malware detector.

To summarize our findings, although TLSH is considered to be robust against random attacks, we proved that it is not robust enough against targeted attacks. We outlined an algorithm to create modifications in order to achieve the highest possible TLSH difference, and tested it on malware samples. We were able to patch 40% of all modifiable samples to the point where even the authors of TLSH consider the scheme broken, while in 56% of the cases we could fool a malware detector built upon TLSH. 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.

ACKNOWLEDGMENTS

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.

REFERENCES

[1] Abdulrahman Alruban, Nathan Clarke, Fudong Li, and Steven Furnell. 2018. 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*. Springer, 135–149.

[2] F. Breitinger, H. Baier, and J. Beckingham. 2012. Security and Implementation Analysis of the Similarity Digest sdhash. In *Proceedings of the 1st International Baltic Conference on Network Security and Forensics (NeSeFo)*.

[3] Donghoon Chang, Mohona Ghosh, Somitra Sanadhya, Monika Singh, and Douglas White. 2019. FbHash: A New Similarity Hashing Scheme for Digital Forensics. *Digital Investigation* 29 (07 2019), S113–S123. https://doi.org/10.1016/j.diin.2019.04.006

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

[5] Jianxing Chen, Romain Fontugne, Akira Kato, and Kensuke Fukuda. 2014. Clustering Spam Campaigns with Fuzzy Hashing. In *Proceedings of the Asian Internet Engineering Conference (AINTEC)*. 66–73. https://doi.org/10.1145/2684793.2684803

[6] E. Cozzi, P.A. Vervier, M. Dell'Amico, Y. Shen, L. Bilge, and D. Balzarotti. 2020. The Tangled Genealogy of IoT Malware. In *Proceedings of the Annual Computer Security Applications Conference*. 1–16. https://doi.org/10.1145/3427228.3427256

[7] Ernesto Damiani, Sabrina De Capitani di Vimercati, Stefano Paraboschi, and Pierangela Samarati. 2004. An Open Digest-based Technique for Spam Detection.. In *PDCS*. Citeseer, 559–564.

[8] Meiqi He, Gongxian Zeng, Jun Zhang, Linru Zhang, Yuechen Chen, and SiuMing Yiu. 2019. 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*. Springer, 248–266.

[9] J. Kornblum. 2006. Identifying Almost Identical Files Using Context Triggered Piecewise Hashing. In *Proceedings of the 6th Annual DFRWS Confenerce*.

[10] Yuping Li, Sathya Chandran Sundaramurthy, Alexandru G. Bardas, Xinming Ou, Doina Caragea, Xin Hu, and Jiyong Jang. 2015. 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.

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

[12] Jonathan Oliver, Chun Cheng, and Yanggui Chen. 2013. TLSH – A Locality Sensitive Hash. In *2013 Fourth Cybercrime and Trustworthy Computing Workshop*. IEEE, Sydney NSW, Australia, 7–13. https://doi.org/10.1109/CTC.2013.9

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

[14] Jonathan Oliver and Josiah Hagen. 2021. Designing the Elements of a Fuzzy Hashing Scheme. In *2021 IEEE 19th International Conference on Embedded and Ubiquitous Computing (EUC)*. IEEE, 1–6.

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

[16] Jayson Pryde, Nestle Angeles, and Sheryl Kareen Carinan. 2018. 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*. Springer, 181–185.

[17] V. Roussev. 2010. Data Fingerprinting with Similarity Digests. In *Research Advances in Digital Forensics VI*. 207–226.

[18] V. Roussev. 2011. An Evaluation of Forensics Similarity Hashes. In *Proceedings of the 11th Annual DFRWS Confenerce*.

[19] Nikolaos Sarantinos, Chafika Benzaid, Omar Arabiat, and A. Al-Nemrat. 2016. Forensic Malware Analysis: The Value of Fuzzy Hashing Algorithms in Identifying Similarities. 1782–1787. https://doi.org/10.1109/TrustCom.2016.0274

[20] József Sándor, Roland Nagy, and Levente Buttyán. 2023. 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)*.

[21] Csongor Tamás, Dorottya Papp, and Levente Buttyán. 2021. SIMBIoTA: Similarity-based Malware Detection on IoT Devices. In *Proceedings of the 6th International Conference on Internet of Things, Big Data and Security (IoTBDS)*. INSTICC, SciTePress, 58–69. https://doi.org/10.5220/0010441500580069