Targeted Attacks against the TLSH Similarity Digest Scheme

Gábor Fuchs*, Roland Nagy† and Levente Buttyán‡
CrySyS Lab, Budapest University of Technology and Economics
Hungary
Email: *gfuchs@crysys.hu, †rnagy@crysys.hu, ‡buttyan@crysys.hu

***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 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 digest 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.**

***Index Terms*—Similarity digest schemes, locality sensitive hashing, targeted attacks, digital forensics, malware detection and classification.**

I. Introduction

SIMILARITY digest schemes (also known as bytewise approximate matching algorithms, similarity digest algorithms, or fuzzy hash algorithms) map an input of arbitrary length to a small size output, such that similar inputs result in similar digest values. These mappings are different from cryptographic hash functions that map even very similar inputs, differing only in a single bit, to completely different hash values. This property of preserving similarity enables similarity digest schemes to measure how different or similar two files are based on their similarity digests alone. Similarity digest schemes, such as Ssdeep [1], Sdhash [2], Nilsimsa [3], and TLSH [4] are used in, for instance, digital forensics [5], spam filtering [6], malware clustering [7], [8], and malware detection [9].

Most applications of similarity digest schemes fall into one of two classes: *fuzzy blacklisting* or *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 [10] and network traffic [11]. 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 digest values. An *anti-whitelisting* attack makes limited modifications to a file that is otherwise rejected based on the whitelist such that the modified file has a digest that is similar to a digest on the whitelist. An ideal version of this attack creates a perfect similarity digest collision. With such an attack, attackers could defeat not only fuzzy whitelisting applications, but essentially *all* uses of the similarity digest scheme.

Robustness of existing similarity digest schemes against such attacks have been extensively studied [5], [12]–[15]. The authors of [14] concluded that Ssdeep and Sdhash are not sufficiently robust for practical use, while TLSH is more difficult to exploit. The authors of [15] investigated how the different similarity digest schemes perform in a variety of binary analysis scenarios, and when trying to match different versions of the same program, they also found that TLSH is the most robust solution.

In this paper, we show that it is rather easy to successfully attack TLSH too. Our threat model is that a knowledgeable attacker knows the exact configuration how TLSH is used in a system or in a forensic process, and modifies input binaries, without necessary access to their source code, and without changing their functionality. They mislead the TLSH scheme in the comparison of these modified inputs to other files in

different ways, resulting in false negative decisions in anti-blacklisting, or false positive decisions in anti-whitelisting applications. The attacker is assumed to have plenty of time to carry out their modifications. We note that making modifications to achieve a precise impact on the similarity digest of the file is actually easier to do in the final binary than its source code, so the constraint of no access to the source code might not make these attacks any harder. As the authors of [15] noted, even small changes in the source code might unexpectedly cause large chain reactions of changes in the output binary due to the complicated mechanics of compilers and alignment requirements.

We propose both an anti-blacklisting and an anti-whitelisting attack against the TLSH scheme fitting the above threat model. Our methods modify executable files (ELF 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 aim to achieve large difference scores returned by the official TLSH difference calculation function for the digests of the modified and the original binaries while minimizing the required modifications 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 [9]. In the anti-whitelisting attack, we achieve a complete collision of the modified binary's TLSH digest to a given arbitrary TLSH digest. 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 digest 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 Section II, we review the related work on the robustness analysis of similarity digest schemes in general, take a closer look at the analysis of TLSH in comparison to a few other schemes presented in [14], and summarize the existing works on targeted attacks against TLSH. In Section III, we introduce the necessary background on the operation of the TLSH similarity digest scheme, and we introduce some details of the ELF file format. In Section IV, we present the basis of our attacks: a framework for modifying ELF binaries without affecting their functionality, and different methods of manipulations to TLSH by adding specially crafted data to the file. In Section V, we present the design of our anti-blacklisting attack, which modifies ELF binaries in a way that the TLSH difference score between the original and the modified binaries becomes high. In Section VI, we discuss the design of our anti-whitelisting attack, explaining how the value of each of the TLSH digest fields can be controlled to match the value of the corresponding field of the target digest, and thus, create the desired digest collision. In Section VII, we evaluate our attacks on a large data set containing malware binaries, and we study how effectively they can be used to evade similarity-based malware detection, as well as how they could be adapted to other configurations and applications of TLSH. Finally, in Section VIII, we conclude our paper.

II. Related work

Ideally, the robustness of a similarity digest scheme should be evaluated with a well-defined notion of similarity in mind that the scheme is supposed to measure. However, the notion of similarity is usually only defined informally or through practical examples stemming from a given application. Some evaluations and comparisons of schemes in the literature [13]–[16] use test inputs generated based on intuitions, such as that a file should stay similar to its original version if only a few bytes are changed, inserted or removed, some larger regions of the file are rearranged, or when even less exactly definable practical changes occur, like software updates or the same source code being compiled with another compiler or compiler settings. The authors of [16] created a framework that automates the performance evaluation of similarity digest schemes by testing them with inputs derived from such intuitive scenarios.

Some works went further and proposed effective targeted attacks against some well-known similarity digest schemes, such as Ssdeep [17], Sdhash [12], and Nilsimsa [3]. In terms of design, the closest ancestor of TLSH among these schemes is Nilsimsa; TLSH can be perceived as its much advanced, more complex, and fine-tuned version. In the analysis of Nilsimsa [3], multiple adversarial possibilities where shown against the scheme, including a targeted (aimed) attack that entailed precise and efficient manipulation of the scheme by taking advantage of a deep understanding of the algorithm.

A. 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. First of all, 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, then the digest scheme can be considered broken.

They made different random modifications in the source codes of some programs without altering their functionality,

and compiled them with the modifications. These modifications included 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[1], 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.

B. Targeted attacks on TLSH

To the best of our knowledge, we are the first to propose efficient targeted attacks on TLSH. The work presented in this paper is a follow-up to our prior work [18], in which we proposed an anti-blacklisting attack on TLSH. The results in this paper are completely new, and do not overlap with those in [18]. More precisely, in this paper, we present new methods for manipulating TLSH values that enable a more efficient anti-blacklisting attack than the one published earlier, as well as a completely new anti-whitelisting attack.

Another paper of ours [19] is logically a follow up to the work presented in this paper, but ended up being published earlier. It points out an easy way to defend against the attacks with our exact current strategy of modifying ELF files, and proposes an extension to the attack framework presented in the current work, which enables concealing the modifications as legitimate machine code of the modified binaries to disable such trivial defense.

III. Background

For a better understanding of our targeted attacks on TLSH, one can benefit from some details on the TLSH scheme itself (Section III-A) and a brief overview of ELF (Section III-B), the binary file format that we use as a practical example input file format to demonstrate our attacks.

A. The TLSH digest calculation algorithm

TLSH is a similarity digest scheme, and as such, it provides two algorithms: the *digest calculation* algorithm calculates the fixed size 35-byte TLSH digest of a file, which preserves its characteristics in a way that the *digest comparison* algorithm can give a score quantifying how different two files are based on their corresponding TLSH digests.

In this section, we discuss the TLSH digest calculation algorithm. Note that its presentation in [4] is somewhat confusing regarding details like byte order. The following is based on the official reference implementation[2]. TLSH has a few alterable parameters; however, they all have default values, which are carefully chosen [20], 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 three 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 were not contained in previous window positions (i.e. the ones containing the last byte of the window). For each window position, these are six new byte triads. With the terminology of k-skip-n-grams[3], from all window positions together they are the 2-skip-3-grams ($n = 3$, $k = 2$) of the whole processed byte stream[4]. Each of these byte triads is hashed to a single byte using the Pearson hash algorithm with an added salt value depending on the form of the given triad. The occurrences of different hash values (throughout all the processed byte triads from all 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*. Default TLSH preserves the bucket counts only for hash values 0-127, discarding the second half of the array. Along with increasing the bucket counts, a single-byte checksum is also calculated.

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 (indexing from 0th) the 31st, 63rd, and 95th ones respectively, in other words, the last value in each of the first three quarters of the sorted version of the remaining array.

The TLSH digest 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 the quartile ratios in percentage modulo 16: `Q1ratio` $= \lfloor 100\frac{q_1}{q_3} \rfloor \bmod 16$ and `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 the above fields 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. Such an input would increase only a few bucket counts, leaving most of the buckets with zero and leading to all quartiles being zero too. Hence, if we tried

[1]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. However, this value is not achievable as an actual difference score between two TLSH hashes, hence, the real maximal value is definitely smaller.

[2]https://github.com/trendmicro/tlsh Last accessed: July 9, 2025

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

[4]Excluding the 2-skip-3-grams that could be formed using the first 4 bytes of the file alone.

calculating the digest header values, specifically the quartile ratios, we would have to divide by zero.

In the digest comparison algorithm, the difference between two TLSH digest values is calculated from the relations of the above described values in the two compared hashes. Each pair of corresponding fields in the two digests has its own possible contribution to the difference, and these contributions are simply summed. For a more detailed review of the TLSH difference score calculation, we refer the reader to [18].

B. Modifying ELF binaries

We demonstrate our attacks against the TLSH scheme on the example of IoT malware samples. As IoT devices often run Linux-based operating systems, the dominant file format among IoT malware samples is the Executable and Linkable Format (ELF), so we have to understand this file format to some extent. Every ELF file starts with a so called *ELF header*, which can reference two, one by one optional header tables, the *program header table* and the *section header table*. The rest of the content of the file can be referenced in different ways by these two header tables. All executable ELF files must contain a program header table. The program header table contains every piece of information required for execution (e.g. what regions of the file to load to what regions of memory before the execution can start). The section header table contains and references metadata that is not required for execution, and is not required to exist at all in executable files.

Our attack framework requires space within the files it can write to, without corrupting them or changing their behavior, and preferably without increasing their size too much. To achieve this, we first remove the reference to the section header table (if there is one), then remove all the parts of the file that are not referenced by the program header table (and not holding bytes of the ELF header or the program header table themselves). The largest region removed this way is usually the end of the file, simply truncating the result; however, it is also possible to have some non-referenced regions in between the ones kept. As described later in detail, our attack framework tries to use (overwrite) these inner regions first to achieve its goal, and proceed to append to the end of the truncated data only if required, possibly making the end result even smaller than the original file. None of these modifications affect the behavior of the program when executed in any way.

IV. Manipulation of TLSH bucket counts

In our attacks, we target manipulating TLSH hashes in particular ways. Apart from `checksum` and `lvalue`, all the other fields of TLSH (the quartile ratio fields `Q1ratio` and `Q2ratio`, and the `codes`) are calculated from the bucket counts. Consequently, manipulation goals on these later fields can be traced back to manipulation goals on the bucket counts. In the following paragraphs we introduce the concept of bucket count goals and the considered types of allowed modifications to the input files, then we propose two different approaches to achieve a set of such bucket count goals by such allowed modifications of the input file. Figure 1 shows an overview of the described framework.

Bucket counts are the 256 counters in the TLSH algorithm, each of which is incremented whenever a selected byte triad is mapped to the corresponding byte value by the Pearson hash algorithm during the processing of the input file. To achieve certain desired changes in the dependent TLSH digest fields during our attacks, we calculate a set of target intervals for the values of each of the 256 bucket counts, which we call *bucket count goals*. These intervals might be bounded, half-bounded, or unbounded by only optionally defining minimum and optionally defining maximum target values. For example, the half-bounded $[100, +\infty)$ interval as a goal for a bucket count b_i means that by the end of the manipulation b_i should be at least 100, while there is no upper limitation to its allowed value.

We consider two types of allowed modifications on input files. First, there might be regions in a file that can be freely overwritten by any equal sized sequence of bytes (e.g. the space between segements loaded to the memory in executable ELF binaries); second, it might be allowed to freely append data to the end of the file (as it is the case with ELF files). We call the overwriteable regions *modification windows*, with the additional abstraction that when it is allowed to append to the end of the file, there is a left-bound (infinite sized) modification window at the end of the file.

By adding a piece of data (i.e. a sequence of bytes) to the file, we increment the vector of bucket counts by the *bucket count contributions* of the given piece of data, which is the vector of total increments to the 256 bucket counts during the processing of the selected byte triads from the piece of data alone; and the *side contributions*, which are the total increments over the byte triads containing bytes from both the piece of data itself and its surrounding context that it is inserted into. Overwriting an existing piece of data subtracts the (own and side) contributions of the original data from the overall bucket counts and adds the contributions of the new data.

During our modifications, we call a set of contributions *neutral* if they do not increase any bucket counts that have upper bounds in the current bucket count goals. Pieces of data with neutral contributions are useful to fill space when we do not wish to increment any more bucket counts. It is worth noting that in the case of default TLSH, the upper 128 bucket counts can always be unbounded in the bucket count goals, as these bucket counts are discarded during the similarity digest calculation. This means that the contributions that increase only these bucket counts are *universally neutral* for any overall attack goal on any file any input file.

A. Preparing the input

The first step is to determine and *claim* the modification windows in the input file. This claiming might entail file format specific changes, like editing header fields. In case of ELF binaries, we remove the references to the section headers, and truncate the file to the content that is actually loaded to the memory during execution as described in Section III-B, then allow appending to the truncated file as well as overwriting the spaces between segments, each with an equal sized piece of data, by defining the corresponding modification windows.

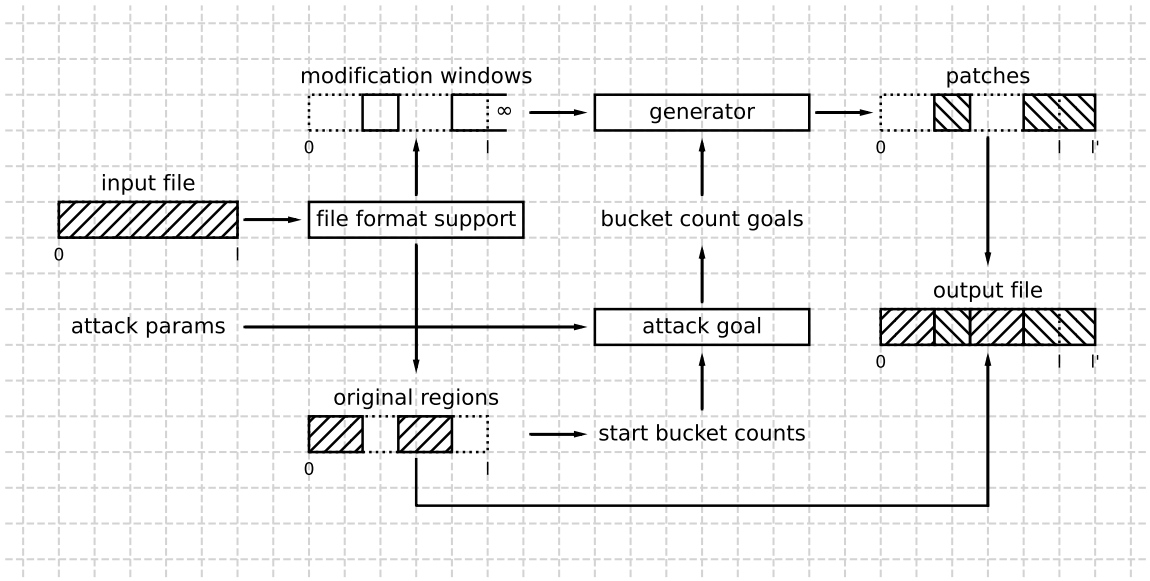

Fig. 1. Overview of the attack framework. Boxes labeled as "file format support", "attack goal", and "generator" represent abstract modules, each with an easily extensible set of separately interchangeable implementations of different concrete strategies. The displayed example of modification windows contains an infinite final window, to which, an adversarial byte sequence of arbitrary length can be written to, thus changing the size of the file: l marks the original length of the file, l' marks the length of the modified file. Note that it is possible that $l' < l$.

Next, we calculate the bucket counts over the file without the modification windows (skipping byte triads that contain any bytes from any of them). We call these resulting counts *start bucket counts*. By filling the modification windows with their new content, we only form new byte triads, resulting in additional bucket count contributions, so every upper bound present in the bucket count goals must be higher than the corresponding start count. For this reason, these start counts are a part of the input to the attack specific algorithm that determines the bucket count goals.

By subtracting the start counts from the bucket count goals (from all present upper and lower bounds), we get what we call *relative bucket count goals* or *bucket count increment bounds*, as these are bounds for the necessary minimal and allowed maximal contributions of the new data written over the modification windows to achieve the original bucket count goals. As from this point we are only increasing bucket counts, from within these increment bounds any negative lower bounds can be dropped, while the presence of any negative upper bounds is an instant failure.

The actual input to either of the algorithms is these increment bounds and the sizes and *contexts* of the modification windows. Context refers to the last few bytes before and first few bytes after the given modification window, which the newly added piece of data will form its side contribution defining byte triads with. It is assumed that the modification windows are independent in the sense that their contexts are made of bytes not belonging to any other modification window.

B. Method I: Greedy generation

This method generates the new content of the modification windows byte by byte, always choosing the momentarily best looking option. How "good" an option for the value of a given byte looks is determined by a heuristic scoring system. The contributions of each of the 256 possible byte values are calculated, which is the sum of the contributions from all newly formed byte triads containing the new byte with its given considered value, and already decided bytes (bytes that have been already decided by this algorithm, or bytes of the context of the modification window). If the contribution of a given byte value exceeds the upper increment bound for any of the buckets, the given byte value is discarded as an option. If the contribution has an increment for a bucket count with a positive lower increment bound, meaning that it is (still) required to increment the given count to achieve the corresponding goal, it is rewarded by its contribution to the given bucket count in score (up to the lower increment bound). This score accumulates over the 256 bucket counts unless the given byte value is discarded in the process. The algorithm collects the non-discarded options to a list sorted in decreasing order by score. The top scoring option is selected, its contributions are subtracted from the increment bounds, and the algorithm moves on to selecting the next byte. If at some point there are no non-discarded options, the algorithm does backtracking by undoing its last choice (and the subtraction of the corresponding contributions) and moving to the next best looking option there, proceeding with that instead. This same backtracking is done if the algorithm runs out of these options for a given byte after stepping back to it multiple times and having tried all the previously listed options. A failure only happens if the algorithm runs out of options for the first byte. The algorithm fills all the finite modification windows (originally the overwriteable regions), and proceeds to generating bytes to the infinite modification window (appending to the file) if it is allowed and there are still positive lower increment bounds.

Note: To reduce computational cost, this method might be improved by brute-forcing blocks of bytes instead of single bytes, selecting the most promising; then either keeping the result and jumping forward to the next entire block, or keeping just the first (few) bytes of the chosen block and sliding the selection in a way that it regenerates both the end of the previous block and some new bytes.

C. Method II: Periodic patterns

Periodic byte patterns in a file have an interesting effect on the TLSH bucket counts. As the same byte pattern repeats, the same byte triads also repeat, resulting in increments of the same few bucket counts again and again. If we can find a pattern that periodically increments a few selected bucket counts, this makes a space efficient and easily scalable way to significantly and precisely increase these few selected bucket counts without affecting others.

We used this principle to achieve the desired bucket count changes in the original form of our anti-blacklisting attack, which is described in detail in [18]. Here we only briefly describe a slightly improved version of the method that we tested in the current generalized framework.

First, the algorithm evaluates all the possible periodic patterns from a predefined pool (currently the periodic patterns with period length 2, like the ASCII string `ABABAB...`), scoring each of them based on their periodically repeating contributions to the bucket counts. The applied scoring function is similar to the one used in the greedy generation method for the selected byte values. A certain number of top-scoring patterns are *preselected*, and passed to a solver, which decides which patterns to use in each of the modification windows and with how many iterations of periodic repetition. The constraints for the solver require that the bucket count goals (lower and upper increment bounds) are fulfilled by the total contributions of the patterns with the chosen numbers of iterations, while they also fit in the modification windows where they are meant to be inserted to. A difference to the original version of the method is that instead of connecting the different applied patterns within a modification window (to each other and the frame of the window) with strings of predefined bytes (previously ASCII `YYYY`), we simply leave gaps of 4 undecided bytes in between, which we fill using the greedy generation method as a final additional step. For this reason we do not need to address the side contributions of the given patterns, neither in the preselection phase, nor in the solver constraints. Although, this can be considered as a hybrid method of using periodic patterns and greedy generation, we still refer to it as the periodic patterns method.

V. Attack I: anti-blacklisting

Making the similar look unsimilar.

In this section, we present our first attack, which aims at modifying arbitrary executable ELF binaries in such a way that

1) their functionality remains unaffected;
2) their size remains unchanged or the size increment is minimal; and
3) the TLSH difference score between their modified and original versions (*self difference score*) becomes as high as possible.

Requirement 2 serves to rule out the exploitation of a known limitation of TLSH, namely that the TLSH value of a file can be easily manipulated by appending large amounts of random bytes or bytes of a benign software to it [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, achieving a self difference score greater than or equal to a given target score, by adding as little unused content to the file as possible.

The anti-blacklisting attack described here is a somewhat generalized but very similar version of our original attack from [18], where we provided a more verbose explanation of the process below.

As every part of the TLSH digest affects the TLSH difference, we have a few choices of fields to manipulate in order to achieve the desired self difference. The `checksum` and `lvalue` fields are not suitable for this purpose: `checksum` can provide only a single point of difference score, while `lvalue` can only be affected by modifying the file size drastically which is ruled out by Requirement 2. This leaves us with the `Q1ratio`, `Q2ratio` and `codes`, which are the fields determined by the bucket counts.

Our chosen strategy is to achieve the required self difference by increasing the q_3 quartile value without changing the q_1 and q_2 values, thus achieving a change in both of the quartile ratio fields sufficient to provide the required difference score, when compared with their original values.

After finding the (smallest) appropriate new $q'_3 \geq q_3$ value, this can be easily translated to bucket count goals, so the attack can be executed in the framework introduced in Section IV: Assuming the use of default TLSH, which keeps only 128 bucket counts; the 33 largest kept bucket counts have to grow to be (or stay) greater than or equal to q'_3 (lower bounds); while the smallest 32 kept bucket counts have to stay less than or equal to q_1, and the next 32 smallest kept counts have to stay less than or equal to q_2 (upper bounds).

One thing to consider when executing this attack in the framework is to achieve the desired self difference relative to the quartile ratio values defined by the bucket counts of the (entire) original file, the bucket counts of the truncated version, or the start counts (see Section IV-A).

VI. Attack II: anti-whitelisting

Making the unsimilar look similar.

In this section, we present our second attack, which aims at modifying binary files in a way that their TLSH digest will be very similar to another arbitrary TLSH digest. While in the previous attack presented in Section V, 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 digest collision, so this is what we aim for.

While this attack usually requires a far greater extent of modification to the files, it is a much stronger attack at the

same time in terms of impact on applications and difficulty to defend against. In the case of the previous anti-blacklisting attack, if the defender knows our method of modification, they can for example proactively blacklist digests of modified samples as well to mitigate the attack. By our 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 rejected (e.g. detected as malware) to make its TLSH digest be the exact same as that of the other, whitelisted sample. With such possible digest collisions, it is impossible to reason for a decision based on the TLSH digests of the files alone. For example, if a malware sample has the exact TLSH digest of a well-known benign executable, it is impossible to detect it as malware based on its TLSH digest 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 digests like SIMBIoTA-ML [22] that make decisions based on TLSH digests without any use of the TLSH difference score calculation.

A. Controlling the digest fields

To modify a file in a way that its TLSH digest will completely match another given digest (*target digest*), we need to be able to precisely control the value of each and every digest field. First, the fields `Q1ratio`, `Q2ratio` and `codes` as they are controlled by the relationships of the bucket counts. Our strategy is to calculate bucket count goals that guarantee the same values for `Q1ratio`, `Q2ratio`, and `codes` as those of the target digest. As we can only increase bucket counts from their current values by addition to the file, each of bucket count goals should enable higher values than the starting value of the given bucket count (so the upper increment bounds will be positive). The `lvalue` and `checksum` fields can be quite easily manipulated as a couple of additional final steps after the common process depicted by Figure 1.

B. Calculating and achieving the bucket count goals

From the `codes` of the target digest, we can exactly see which buckets should belong to which quartile intervals. From this information, we can set a few lower bounds for our target quartile values, as by adding data to the file, we can only increase the current bucket counts and, thus, the resulting quartile values. If a bucket count is currently b_i and $\texttt{code}_i = 0$ in the target digest (i.e. $b_i \leq q_1$), we know that our new q_1 has to be greater than or equal to 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:

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

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

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

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 digest, which are $\bmod 16$ stored values of $r_1 = \lfloor \frac{q_1}{q_3} \cdot 100 \rfloor$ and $r_2 = \lfloor \frac{q_2}{q_3} \cdot 100 \rfloor$, respectively. Let's define functions dqr_1 and dqr_2 to map the ratio of a pair of quartile value candidates to the difference of the corresponding quartile ratio field in the target digest and their current encoded ratio:

$$\text{dqr}_1 : \frac{q_1}{q_3} \mapsto (\texttt{Q1ratio}_{\text{target}} - \lfloor \frac{q_1}{q_3} \cdot 100 \rfloor + 8) \bmod 16 - 8 \tag{4}$$

$$\text{dqr}_2 : \frac{q_2}{q_3} \mapsto (\texttt{Q2ratio}_{\text{target}} - \lfloor \frac{q_2}{q_3} \cdot 100 \rfloor + 8) \bmod 16 - 8 \tag{5}$$

The value of each of these function hints in which direction the corresponding current ratio needs to be adjusted to achieve its target value by minimal change to the planned quartiles themselves. This is useful because, while we can only increase the quartile values, this still enables both increasing and decreasing their ratios.

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 cause the ratio to change by more than one percent, resulting in skipping the ratio value that would result in the desired `Q1ratio` or `Q2ratio` value. As each modification increments one of the quartile values, it will cease this issue sooner or later.

Algorithm 1 Achieving the target quartile ratios with planned quartile values

```
q1 ← q1,start
q2 ← q2,start
q3 ← q3,start
while dqr1(q1/q3) ≠ 0 or dqr2(q2/q3) ≠ 0 do
    while dqr1(q1/q3) < 0 or dqr2(q2/q3) < 0 do
        q3 ← q3 + 1
    end while
    while dqr1(q1/q3) > 0 do
        q1 ← q1 + 1
    end while
    while dqr2(q2/q3) > 0 do
        q2 ← q2 + 1
    end while
end while
return q1, q2, q3
```

It is still not guaranteed that $q_1 \leq q_2 \leq q_3$ holds for the resulting values; however, in all practical cases we encountered so far in our experiments, this was the case. Thus, even though the algorithm could be adjusted to address this issue, our implementation of the attack simply contains an assertion for this to be true.

Having calculated the appropriate q_1, q_2, q_3 quartile values, we can set the bucket count goals:

$$\texttt{bc\_bound}_i = \begin{cases} [0, q_1] & \text{if } \texttt{code}_{i,\text{target}} = 0 \\ (q_1, q_2] & \text{if } \texttt{code}_{i,\text{target}} = 1 \\ (q_2, q_3] & \text{if } \texttt{code}_{i,\text{target}} = 2 \\ (q_3, \text{inf}) & \text{if } \texttt{code}_{i,\text{target}} = 3 \end{cases} \tag{6}$$

An additional 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 $\texttt{bc\_bound}_i$ goal 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 when many bucket counts share the same value around the transition between the quartile itervals. For example, if in the ascending (i.e. sorted) order of the 128 kept bucket counts (indexing from 0th) the 31st count equals the 32nd and 33rd, they will all map to 0 in `codes`, because even though the 32nd and 33rd are past the 31st count, which defines q_1, their values are still less than or equal to q_1. If we have this phenomenon in the target digest, 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.

Having determined the bucket count goal ($\texttt{bc\_bound}_i$) for each bucket count we want to achieve, we can use one of the strategies described in Section IV to actually achieve matching bucket counts by adding content to the modification windows. Similarly to how we conducted our first attack in Section V, we use the bucket counts of the file without the content of the modification windows as a starting point.

C. Achieving the required file size

By achieving the previously calculated bucket count goals, we guaranteed the required values for `Q1ratio`, `Q2ratio` and `codes`. The `lvalue` field depends only on the file size. While we cannot make the file smaller by appending to the end, which is the only way we support modifying the file size, we can easily make it bigger. The only challenge is to do so without changing the previously set bucket count relations. This can be solved easily: we append some data with neutral contributions and with appropriate length to reach the required `lvalue`. A simple way to do this is to use a neutral periodic pattern (see Section IV-C). While the inner byte triads of the neutral pattern are guaranteed not to contribute to any bucket counts used for the digest 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 issue, we do a breath first search (BFS) of all possible continuations on the byte level to insert at the end of the data until we find the shortest continuation that transitions to any neutral periodic pattern. Once we found such a "bridge", we can append any number of iterations of the neutral pattern, and thus achieve the required file size.

D. Faking the checksum

If we followed through and succeeded with all the previous steps above, we are guaranteed to have an almost perfect digest collision. The only digest field that is not guaranteed to match the corresponding value of the target digest is the `checksum`. This means that the TLSH difference between the digest of our modified file and the target digest 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. 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 VI-C instead, to find the shortest sequence of bytes to append that changes the checksum to its target value while, at the same time, keeping all the bucket counts under their upper bounds in the bucket count goals.

If we have succeeded with all the steps above, then we should have a complete TLSH digest collision.

VII. Evaluation

We evaluate our attacks by performing them on samples of an IoT malware dataset. First, we measure the proportional amount of data required to be inserted in the files being patched by each attack in order to achieve their corresponding goals. The choice of patching IoT malware binaries further enables testing our attacks as adversarial example creation strategies against SIMBIoTA, a similarity based malware detection solution using TLSH.

A. SIMBIoTA

SIMBIoTA [9] is a malware detection solution for IoT devices. Malware detection on IoT devices is challenging, because IoT devices have very limited resources regarding computational power, memory, and storage, so they do not meet the resource 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 digest and difference score.

The computation done on the IoT device to classify a binary as malware or benign is as simple as calculating its TLSH digest, which is then compared to a carefully chosen relatively small set of TLSH digests for known malware samples, called *dominating set*. This set of digests is regularly updated from an antivirus server, and is guaranteed to contain at least one digest for every relevant malware sample the server has ever seen that it is considered to be similar to. A new binary is classified as malware if its TLSH difference from one of the digests in the dominating set 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].

B. Data set

The data set used for the evaluation of our attacks is the *CrySyS-Ukatemi benchmark data set of IoT malware 2021* or CUBE-MALIoT-2021[5]. This data set is publicly available for research and education purposes, and consists of 29,209 ARM and 18,715 MIPS malicious ELF binaries (malware samples). For their already great number, we decided to work with the ARM samples only.

[5]https://github.com/CrySyS/CUBE-MALIoT-2021 Last accessed: July 9, 2025

C. The common framework

All of our attacks start by parsing the ELF executable, removing the reference to its section header table by editing the corresponding values in its ELF header, and removing all the data from the end of the file that are not referenced by any program header entries by truncating its content to the end of the last segment within the file (or the end of the program header table if it happens to be after the last segment). From the 29,209 ARM samples of the data set, 830 samples contained program header entries that referenced parts of the file after the end of the file. To our knowledge, such ELF files are invalid, so we decided to ignore them, and worked with the remaining 28,379 samples.

We further mark all windows within the truncated file that do not belong to any of the ELF header, the program header table or the segments referenced by the program header entries. All of our attacks first overwrite these inner windows, and then proceed to appending to the truncated file if necessary.

We collected the original size, the truncated size, the sizes of the inner modification windows, and the patched size for each successfully patched sample, and derived the following values from these data:

- Total size of original data: truncated size minus the total size of inner modification windows;
- Total size of added data: the total size of inner modification windows, plus the difference of the patched size and truncated size (i.e. the amount of appended data);
- Modification ratio: the ratio of total size of added data over the total size of original data.

Among the remaining 28,379 samples, which we did not consider invalid, the truncation decreased the file sizes by 17.4% on average. There were 352 samples that could not be truncated at all, as the end of the file was referenced in program header entries (e.g. the entire file was loaded by a single load entry), and there were many that became smaller by only a few tens of bytes down to a 0.02% decrement in size. The biggest decrement of 92.2% was unique to a single sample, followed by many samples with similar decrements down from 59.2%.

No samples contained more than one inner modification windows and 17,497 (61.7%) samples contained a single inner window with an average proportional size of 0.64% of the truncated file size. In 7600 samples, the size of this single inner window was only 4 bytes, which was the smallest window size. There were some more samples with similarly small inner windows (i.e., 6, 8, 12, ... bytes), but in much smaller numbers. Excluding only the samples with 4 bytes long inner modification windows, for the remaining 9897 samples with bigger inner modification windows (34.9% of all valid samples) the average proportional size of the window jumped to 1.12% of the truncated file size, the largest such proportional size being 10.6%.

We do not report detailed measurements of the practical runtime efficiency of our attacks, because, when successful, they all run within a few seconds, and we were able to conduct *all* of our evaluation tests involving 28,379 samples and many attack configurations in only a few hours. It seems unlikely in practice that an attacker would ever need to modify such large amount of samples at once, and even if that was the case, spending a couple of hours on patching thousands of samples does not seem to be a serious limiting factor for any attacker.

D. Anti-blacklisting

We tested our anti-blacklisting attack with three different target self difference scores:

- 40 (48) – the detection threshold of SIMBIoTA
- 86 (96) – the threshold proposed for ELF binaries in [14] by the authors of TLSH (see Section II-A)
- 144 (144) – a higher setting that is still practically achievable with the current algorithm

In parentheses are the actual TLSH self difference scores guaranteed through the target differences in the quartile ratio fields, which are multiples of 12 for significant differences.

For all three of these different settings we tried both the greedy generation algorithm (Section IV-B) and the algorithm using periodic patterns (Section IV-C).

We take the initial quartile ratio values from the truncated version of the file rather than from the original version, and thus guarantee a self difference score from the truncated version. The reason for this is that the truncation alone might cause a huge TLSH difference from the original version (i.e., 78.8 on average and 327 at maximum among the valid ARM samples), which has nothing to do with the efficiency of our attack on TLSH.

The outcomes of the attacks on all the ARM samples of the data set are summarized in Table I.

The higher 144 setting was chosen because the highest (156, 168) settings cannot be practically achieved by only increasing the q_3 value. Such an attack could be possible by manipulating the quartile values similarly to how it is done in our anti-whitelisting attack (see Section VI-B). This setting of 144 already fails in some cases (see "Failed reaching quartile relations" in Table I).

Another reason of failure was generator failures. For greedy generation this would mean that the backtracking algorithm ran out of possibilities, which did not happen with any samples on the three settings. In case of the periodic pattern method, this means that the solver proved that the bucket count goals cannot be fulfilled with the preselected pool of patterns, which also only happened with very few samples on the higher settings.

We divide the successfully patched samples for each configuration into two groups: the ones that did not grow in size compared to their initial truncated form (*nongrowing*) and the ones that did (*growing*). The nongrowing samples were patched using their inner modification windows only, not providing exact data on how much space the given attack required.

In case of the growing samples, we assume that the attack required the space of the inner modification windows and the appended bytes together, which is the "total size of added data" defined earlier. This assumption seems to work well for greedy generation, but not for the periodic pattern solution. Unfortunately, we could not find a way to guide the solver

Target self difference score			40		86		144	
Generation method			greedy	patterns	greedy	patterns	greedy	patterns
Invalid samples			830					
Failed	Failed reaching quartile relations		0		1		123	
	Generator failed		0	0	0	1	0	2
Succeeded	Nongrowing		8,858	11	7,207	0	5,522	0
	Growing	$\leq 20\%$ modification ratio	19,521	28,228	21,171	28,238	22,459	27,976
		$> 20\%$ modification ratio	0	140	0	139	275	278

TABLE I
PARTITION OF THE 29,209 ARM SAMPLES AMONG DIFFERENT RESULTS OF EACH CONFIGURATION OF THE ANTI-BLACKLISTING ATTACK

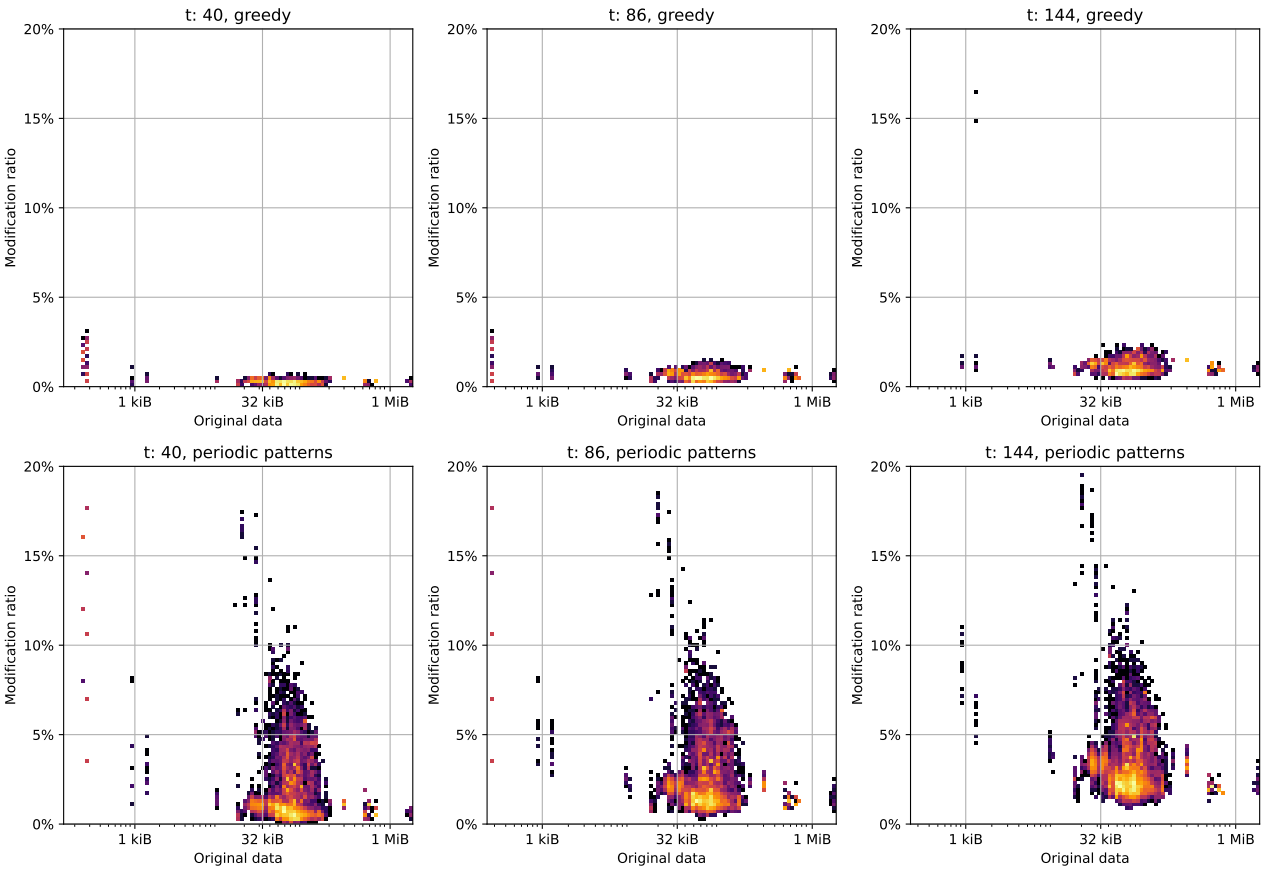

Fig. 2. Modification ratios of the successfully patched malware samples that have grown in size compared to their truncated form in different settings of the anti-blacklisting attack. The graphs are heat maps visualizing distribution: brighter colors represent more samples, each color represents an interval on a logarithmic scale. Some outlier values (above 20%) are cut off especially at the higher settings.

to always use up the inner modification windows before appending to the end of the file without drastically slowing it down. We could only revert the slowdown by decreasing the number of preselected patterns from the current 128 to around 16, which in turn made the solver fail in a large number of cases. Regardless the configuration, the periodic pattern method seems to give worse results than the much simpler and faster greedy generation.

Figure 2 shows the modification ratios of growing samples for both generation methods on the three target settings. For greedy generation the average modification ratios of growing samples were 0.26%, 0.59%, and 1.38% on the three settings, while for periodic patterns they were 1.63%, 2.29%, and 3.97%.

1) Comparison to existing research: As reviewed in Section II-A, the robustness of TLSH (along with other similarity digest schemes) 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 unnecessary parts 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 the TLSH digest and difference calculations.

Their proposed similarity detection threshold for executable ELF binaries is the TLSH difference 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 greater than or equal to 86 points from its original version (i.e. self difference) (see Section 5 of [14]).

Our attack guarantees TLSH self difference through the changes to the quartile ratio fields alone, possibly changing other values in the process, which further increases the actual achieved self difference.

While in the attack we aim for the next multiple of 12, which is 96 in case of the 86 points target self difference score, in our earlier experiments [18] most binaries patched with the lower setting of 84 points target self difference also exceeded 86 points in actually achieved self difference. As shown earlier, even the higher 96 difference could be guaranteed by an addition of 0.59% of the original data on average. Considering all successfully patched samples, including the nongrowing ones, the average growth proportional to the truncated size was 0.39% with the greedy generation algorithm. Considering the entire original files before truncation, the sizes of the patched files were 82.9% of the original file size on average, which means that there were a lot more space for manipulations even without increasing the file size in most practical cases.

2) Adversarial examples to SIMBIoTA: If we were to create adversarial examples to SIMBIoTA by patching malware samples in a way that the modified versions are guaranteed to evade detection by SIMBIoTA, we would have to guarantee that the TLSH digests of the modified versions are not similar (i.e. have a difference score above 40) to any of the digests 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 [23]. If the modified version is very different from the original version, then it is likely to be outside of its cluster and dissimilar to any samples within, while it is also unlikely to become similar to a sample from another cluster.

Consider a version of SIMBIoTA that has seen all the ARM samples of CUBE-MALIoT-2021. To simulate the strongest possible version of the antivirus, we use the entire set as a dominating set. As we conducted our tests on the samples of the same data set, the dominating set contains all of the original versions of our patched samples.

Similarly to how we started out from the truncated form of the binaries in the evaluation of our attacks, we simulate SIMBIoTA as if it has seen the truncated versions of the 28,379 ARM malware binaries that are considered valid instead of the original 29,209 samples.

Let's evaluate our method as a tool to create adversarial examples to this simulated version of SIMBIoTA. Among the samples successfully patched with the greedy algorithm on the target self difference setting of 40 (effective 48), the lowest self difference was actually 48. However some much lower difference scores occurred to other samples in the data set of truncated binaries, the smallest such difference score being only 5 points. 72.4% of these patched samples had original truncated binaries with difference scores of 40 or below, which means that the simulated SIMBIoTA still classifies them as malware, and only the other 27.6% evades detection. Among the samples patched with the periodic pattern algorithm, the smallest difference was similarly 5 points of difference score, and the detection rate was 72.1%. On the 86 (effective 96) setting, the samples patched with greedy generation achieved a 8.82% detection rate with the smallest difference score still being 5; while the periodic pattern samples had a 8.06% detection rate and 9 points as the smallest difference score. Finally, on the 144 setting, with both greedy generation and periodic patterns the smallest difference score was 27, and only 14 (0.05%) of the 28,256 successfully patched samples were detected.

The explanation to the above data is that by gradually making a sample more and more different from its original form, it is possible that first we are making it more similar to other samples within the same cluster. Only when the difference grows larger, it becomes more and more probable that the modified sample has left the cluster of the original sample, while at the same time, it is still unlikely that it enters another cluster.

We conclude that our attacks on TLSH are efficient methods to create adversarial examples to SIMBIoTA, and that an attacker can use the target setting to achieve a large likelihood of evading detection even without knowing the dominating set that SIMBIoTA works with.

E. Anti-whitelisting

We evaluate our anti-whitelisting attack by trying to patch the 28,379 valid samples to all have the same TLSH digest: the digest of a well known benign executable.

1) Selecting the target digest: Due to the limitation of only being able to increase file sizes in most stages of patching, and thus being able to mutate files to perfectly match the digests (more specifically the `lvalue`) of larger files only (see Section VI-C), we chose a common target digest of a file larger than almost all malware samples. Note that the initial truncation of the patched binaries might help in some cases, as the target has to be larger than the truncated form (or even more specifically the form where the bucket count goals are already achieved). This chosen target was the digest of a version of the GCC compiler executable of the same 32-bit ARM architecture the malware samples are for, which is just over a megabyte in size, larger than 28,342 (99.87%) of the 28,379 valid ARM malware samples in the data set. The remaining 37 samples were significantly larger, and stayed

larger even after truncation. The selected version of GCC 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 correctly without unacceptable false positives will be impossible based on their TLSH digests alone.

2) The attack: To have more meaningful results, we split the attack into two phases: first faking the bucket count related values (`codes`, `Q1ratio`, and `Q2ratio`); and then faking the `lvalue` and `checksum` too. This way we can capture how much addition is required to fake every field but the `lvalue` (and the `checksum` which takes appending only a few bytes). In a realistic attack, we might not even care about the `lvalue` and the `checksum`. But even if we always wanted perfect collisions including these fields, this two-phase approach is still meaningful, because the size increments in the first phase show how much bigger the target file should be than the truncated form of the patched file for probable success. So instead of a single "patched size" value introduced in Section VII-C, we collect the resulting file size after both phases.

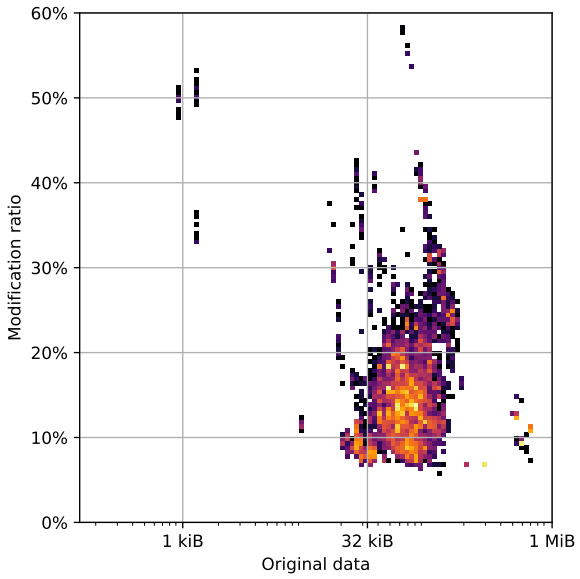

Fig. 3. Modification ratios of the successfully patched malware samples that have grown in size compared to their truncated form in the first phase anti-whitelisting attack, achieving collisions in all of the bucket count related TLSH digest fields to corresponding values in the digest of the GCC executable. The graph is a heat map visualizing distribution: brighter colors represent more samples, each color represents an interval on a logarithmic scale.

3) Results with greedy generation: We successfully patched all the valid ARM binaries from the data set, except for the 37 that was larger than GCC. These larger binaries could be also patched in the first phase to achieve a near perfect digest collision.

Figure 3 showcases the first phase modification ratios for the successfully patched samples that have grown in size compared to their truncated form. As in the evaluation of the anti-blacklisting attack, this selection shows the efficiency of the attack without depending on the presence or size of inner modification windows. This excludes only 1 sample of the 28,342, which was patched in the first phase using only its inner modification window.

Other than in the cases of really small samples and some other exceptions, the required modification ratio tends to be around 10-20% (10.7% and 16.2% being the quartiles), which is roughly 10 times more than what we usually modified in the anti-blacklisting attack.

After executing the second phase of the attack as well for each sample, all 28,342 samples (99.9%) 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 digest of GCC.

4) Periodic patterns for anti-whitelisting: The periodic pattern generation method described in Section IV-C was designed specially for the anti-blacklisting attack, where only a few bucket counts need to be increased. When requiring this method to modify most of the 128 bucket counts as it is required in the anti-whitelisting attack, the fixed number of top scoring periodic patterns do not cover the required contributions, and the patching fails. As offering more patterns to the solver is impractical due to drastically slowing it down, the solution could be to improve the preselection algorithm to consider the possible contributions of the pool of patterns together.

We briefly experimented with an alternative way of selecting the periodic patterns one by one and subtracting the contributions of the determined count of iterations from the increments bounds before moving on to select the next pattern somewhat similarly to how the greedy generation algorithm selects the bytes. However, the results were consistently worse than that of the simple greedy generation.

It might also be possible to construct patterns without going over all the possible ones, maybe enabling the selection of patterns with longer period lengths as well.

F. Discussion

We presented and evaluated our attacks against fuzzy blacklisting and fuzzy whitelisting of ELF binaries with default TLSH; however, they should interfere with other applications as well. The anti-blacklisting attack should work against clustering and closest match search, while the anti-whitelisting attack should work against all possible applications where the required modifications are possible on the data TLSH is calculated on. 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 attacks against other configurations.

1) Other configurations of TLSH: There are a couple of variables in the configuration of TLSH; however, as stated in Section III-A, practically only the defaults are used. In spite of this, it is interesting to look into how these settings would affect our attack. The two officially supported[6] adjustable settings in the build-time configuration of the reference implementation are checksum size (1 or 3 bytes), and number of

[6]https://github.com/trendmicro/tlsh#building-tlsh Last accessed: July 9, 2025

buckets (128 or 256), with the first option being default for both.

The checksum in TLSH is an attempt to identify completely identical files. This field is irrelevant in the anti-blacklisting attack as it can contribute only a single point of TLSH difference score. In the anti-whitelisting attack we manipulate the file content to match a target checksum value, which can usually be done by changing a single byte. Using a bit longer checksum (e.g. 3 bytes) makes this a bit more difficult, but it should still be achievable by changing a few bytes only. In practice this would mean a few more steps of depth in the BFS described in SectionVI-D.

The number of buckets used is more interesting. With the upper 128 buckets being discarded by default, they always have the $[0, +\infty)$ bucket count goal regardless the goal of the attack, which makes the contributions to them universally neutral. Keeping all 256 buckets would disable these universally neutral contributions, decreasing the usual number of bucket counts without upper bounds from $128+32 = 160$ (the discarded plus the ones over q_3) in the bucket count goals to 64 (the ones over q_3 among the 256). This could make both the greedy generation and the periodic pattern approaches more difficult, as well as finding a neutral periodic pattern to blow up the file size in the anti-whitelisting attack. An interesting approach would be to already consider the target file size while determining the bucket count goals, as by keeping all the 256 bucket counts, their sum will always be $6(l-4)$, where l is the size of the file ($l-4$ being the number of sliding window positions, each selecting 6 byte triads).

2) Other applications of TLSH: We showed that ELF binaries can be patched in a way that the result has a high TLSH difference from the original version or even have an (almost) identical TLSH digest to an arbitrary given TLSH digest, without drastically increasing the file size. What enabled this attack was the fact that an ELF binary has continuous regions that the attacker can overwrite or even append data to the end of the file without affecting the behavior of the program. Patching any other type of file with this same method has the same requirements. In practice, some file formats may enable appending bytes to the end of the file which are later ignored, adding metadata that is never validated, adding arbitrary comments, or using padding between regions to keep alignment. Also note that both the greedy and the periodic generation techniques might be successful even if the allowed bytes or byte sequences are limited (e.g. to printable ASCII or alphanumerical characters).

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 example of these is the system described in [24], 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 [25], where accesses to confidential documents are logged along with the TLSH digests of the accessed documents, for in case of document leakage the perpetrator can be more easily identified through comparing the digest 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. Both of these applications are somewhat similar to fuzzy blacklisting. Neither of the two 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 kinds of editing our attacks are requiring.

VIII. Conclusion

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

The attacks were implemented in a modular framework. The first pluggable component (file format support) adds support to create or identify modifiable regions within input files of a given format, which were ELF executable binaries in our examples. The second such abstract component (attack goal) implements the translation of an abstract attack goal to a unified representation used by the last such component of the framework. We added two implementations for this: one for carrying out an anti-blacklisting attack with a configurable target TLSH difference score to be achieved between the original and modified versions of the input file, and another for carrying out an anti-whitelisting attack with a configurable target TLSH digest that the modified file should produce exactly or nearly exactly. The third abstract component (generator) takes the output of the previous components, and generates patches to the identified modifiable regions in a way that guarantees to achieve the attack goal encoded by the unified representation provided by the attack goal module. We evaluated two different generation strategies, which ended in the conclusion that the simpler one of the two (greedy generation) is not only faster, but also more effective. In the end the framework applies the generated patches to the input file and creates the output, which, in case of a successful attack, is guaranteed to have a TLSH digest with the desired properties. The anti-whitelisting attack optionally takes a few more steps appending bytes to the end of the binary to also match the length and checksum values of the target TLSH digest.

As all combinations of the above modules proved to be quite efficient in terms of time complexity, the main points of our evaluation were the success rates of the different configurations, and the proportional amount of modifications they have done to the input files to achieve their goals.

The anti-blacklisting attack, which affects the security of most applications of TLSH had a large success rate, only failing less than 0.5% of the cases even on the highest tested setting guaranteeing the TLSH difference score 144 between the original and modified binaries. All setting of the attack required less than 1.5% of modifications compared to the size of the files. We tested the successfully patched samples to see whether they can evade the detection by a strong simulated

version of SIMBIoTA, a TLSH-based antivirus solution, that has seen the entire dataset of original files. The detection rate on the modified samples with the highest difference setting was 0.05%, compared to the original 90% detection rate of SIMBIoTA on previously unknown malware.

We evaluated the anti-whitelisting attack, which affects the security of *all* possible applications of TLSH, by patching samples to match the TLSH digest of the executable binary of the `gcc` compiler of the same architecture. The main phase of the attack (without the final append) succeeded for all input binaries and required around 16% of modification on average (under 21% in 90% of cases). The additional phase of matching the file size described by the target digest to achieve complete digest collision is only possible if the so far modified file is smaller than the target size. As `gcc` is quite large compared to most malware samples in our dataset, we were able to achieve complete TLSH digest collision with the `gcc` binary for 99.87% of the malware samples.

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

Finally, we note that TLSH is used by VirusTotal, the largest malware repository in the world, for similarity-based searches, therefore, we notified them about our attacks.

ACKNOWLEDGMENTS

The research presented in this paper was carried out in the context of the European Union project no. RRF-2.3.1-21-2022-00004 within the framework of the Artificial Intelligence National Laboratory and in the project no. 2023-1.1.1-PIACI_FÓKUSZ-2024-00030 funded by the National Research, Development and Innovation Office of Hungary. The presented work also builds on results of the project no. 2018-1.2.1-NKP-2018-00004 (SETIT Project), which was implemented with the support provided from the National Research, Development and Innovation Fund of Hungary.

REFERENCES

[1] J. Kornblum, "Identifying almost identical files using context triggered piecewise hashing," in *Proceedings of the 6th Annual DFRWS Conference*, 2006.

[2] V. Roussev, "Data fingerprinting with similarity digests," in *Research Advances in Digital Forensics VI*, 2010, pp. 207–226.

[3] E. Damiani, S. D. C. di Vimercati, S. Paraboschi, and P. Samarati, "An open digest-based technique for spam detection." in *PDCS*. Citeseer, 2004, pp. 559–564.

[4] J. Oliver, C. Cheng, and Y. Chen, "TLSH – A Locality Sensitive Hash," in *2013 Fourth Cybercrime and Trustworthy Computing Workshop*. Sydney NSW, Australia: IEEE, Nov. 2013, pp. 7–13.

[5] V. Roussev, "An evaluation of forensic similarity hashes," *Digital Investigation*, vol. 8, pp. S34–S41, 2011, the Proceedings of the Eleventh Annual DFRWS Conference.

[6] J. Chen, R. Fontugne, A. Kato, and K. Fukuda, "Clustering spam campaigns with fuzzy hashing," in *Proceedings of the Asian Internet Engineering Conference (AINTEC)*, 11 2014, pp. 66–73.

[7] N. Sarantinos, C. Benzaid, O. Arabiat, and A. Al-Nemrat, "Forensic malware analysis: The value of fuzzy hashing algorithms in identifying similarities," in *2016 IEEE Trustcom/BigDataSE/ISPA*, 08 2016, pp. 1782–1787.

[8] Y. Li, S. C. Sundaramurthy, A. G. Bardas, X. Ou, D. Caragea, X. Hu, and J. 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.

[9] C. Tamás, D. Papp, and L. 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)*, INSTICC. SciTePress, 2021, pp. 58–69.

[10] J. Pryde, N. Angeles, and S. K. 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*. Springer, 2018, pp. 181–185.

[11] B. Charyyev and M. H. Gunes, "Detecting anomalous iot traffic flow with locality sensitive hashes," in *GLOBECOM 2020-2020 IEEE Global Communications Conference*. IEEE, 2020, pp. 1–6.

[12] 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.

[13] M. Martín-Pérez, R. J. Rodríguez, and F. Breitinger, "Bringing order to approximate matching: Classification and attacks on similarity digest algorithms," *Forensic Science International: Digital Investigation*, vol. 36, p. 301120, 2021.

[14] J. Oliver, S. Forman, and C. Cheng, "Using randomization to attack similarity digests," in *International Conference on Applications and Techniques in Information Security*. Springer, 2014, pp. 199–210.

[15] F. Pagani, M. Dell'Amico, and D. Balzarotti, "Beyond precision and recall: Understanding uses (and misuses) of similarity hashes in binary analysis," in *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*, ser. CODASPY '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 354–365.

[16] T. Göbel, F. Uhlig, H. Baier, and F. Breitinger, "Frasher – a framework for automated evaluation of similarity hashing," *Forensic Science International: Digital Investigation*, vol. 42, p. 301407, 2022, proceedings of the Twenty-Second Annual DFRWS USA. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2666281722000889

[17] H. Baier and F. Breitinger, "Security aspects of piecewise hashing in computer forensics," in *2011 Sixth International Conference on IT Security Incident Management and IT Forensics*, 2011, pp. 21–36.

[18] G. Fuchs, R. Nagy, and L. Buttyán, "A practical attack on the TLSH similarity digest scheme," in *Proceedings of the 18th International Conference on Availability, Reliability and Security*, ser. ARES '23. New York, NY, USA: Association for Computing Machinery, 2023.

[19] G. Fuchs, T. Fülöp, and R. Nagy, "Concealing targeted attacks on the tlsh similarity digest scheme," in *Proceedings of the Digital Forensics Doctoral Symposium*, 2025, pp. 1–7.

[20] J. Oliver and J. Hagen, "Designing the elements of a fuzzy hashing scheme," in *2021 IEEE 19th International Conference on Embedded and Ubiquitous Computing (EUC)*. IEEE, 2021, pp. 1–6.

[21] J. Sándor, R. Nagy, and L. 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] D. Papp, G. Ács, R. Nagy, and L. Buttyán, "SIMBIoTA-ML: Lightweight, 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)*, INSTICC. SciTePress, 2022, pp. 55–66.

[23] E. Cozzi, P. 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*, 2020, pp. 1–16.

[24] M. He, G. Zeng, J. Zhang, L. Zhang, Y. Chen, and S. 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*. Springer, 2019, pp. 248–266.

[25] A. Alruban, N. Clarke, F. Li, and S. 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*. Springer, 2018, pp. 135–149.