TC-NetTrack: an Approach for Creating Digital Evidences for Flows in IP Networks

TC-NetTrack: An Approach for Creating Digital Evidences for Flows in IP Networks

Diana Gratiela Berbecaru
Department of Control and Computer Engineering
Politecnico di Torino, Italy
diana.berbecaru@polito.it

***Abstract*—Effective network forensics analysis necessitates the capability to reproduce any network event, irrespective of the user's motivation, the characteristics of earlier events, and whether the origin of the events was an unauthorized intruder or a legitimate user. Numerous packet analyzers, network monitoring systems, and intrusion detection tools today enable network measurement and analysis through packet capturing, which facilitates the examination and reconstruction of network events and user activities. However, they lack the ability to offer cryptographic proof for the packet flows that have been captured. Building on this motivation, we provide the initial steps toward the development of TC-NetTrack: a device that utilizes trusted computing to generate digital proof for specific packet flows captured either on a node's network interface or across a network link. Since TC-NetTrack needs to produce evidence that holds probative value, the most suitable method to achieve this is through digital signatures. However, signing each packet individually is not practical or efficient; therefore, we adopted a method known as tree chaining. We implemented this using Merkle trees alongside an optimized algorithm for traversing the tree, which conserves both space and time when the signer generates packet flow evidence. We assessed the tree chaining technique on three different platforms, each featuring varying processors and memory capacities. TC-NetTrack could be beneficial in application scenarios or use cases that demand verification of the recorded packet flows, including stock trading, financial applications, attack analysis, or military operations.**

***Index Terms*—digital evidence, network monitoring, Merkle trees**

I. Introduction

The growing reliance on 'untrusted' Internet architecture and protocols for even sensitive operations compels security experts and law enforcement agencies to develop systems that improve the comprehension of network incidents. Additionally, there is a rising necessity to produce digital evidence for events that can be presented in court, particularly in situations that require probative value, such as investigations of cyberattacks, stock exchanges, or financial applications. Digital evidence refers to information of probative significance that is stored or transmitted in binary format. Examples of digital evidences include application and system files like tests, images, log files, or data that is not typically utilized by operating systems but is saved on disks. In this study, our objective is to generate digital evidence for packet flows, representing information with probative value regarding the traffic observed on a network node interface, for instance, a router or an application server, or along a network link.

Despite the availability of numerous hardware and software solutions for packet capturing and analysis, including tools like Wireshark [1], Snort, Suricata, and Tcpdump, none offer cryptographically secure digital evidence of network traffic [2]. Network packets store information about: a) device-related details, such as IP address, geolocation at a specific time, and manufacturer; b) user-related information, including time of user login, websites accessed, (list of) files downloaded, duration of user session, and social media activity; c) payload data, such as email content, files downloaded, email attachments, images, and data transferred. Information obtained from packet flows can aid in tracing attacks like Denial of Service (DoS), uncover suspicious behavior, identify both successful and failed login attempts, or unauthorized file downloads. Furthermore, packet flow data can be leveraged to explore data breaches or intellectual property theft, detect malware infections, track visited websites, and reconstruct files, documents, or emails sent over the network. Packets can be examined individually or collectively, in the form of client-server exchanges, packet streams, flows, or sessions. Although a significant portion of modern traffic is encrypted, certain details in the packets remain in clear, such as the sender and receiver IP addresses and various metadata. As a result, indirect information inferred from multiple network packets may potentially serve as evidence, albeit lacking cryptographic protection, when investigating cyberattacks. For example, a large stream of ICMP packets dispatched from one host to another within a brief time period may suggest a DoS attack. Some research suggests that utilizing packets as evidence may be problematic because IP packets can be spoofed [3], IP addresses may change often in dynamic environments, and frequently they cannot be linked directly to a specific individual. Conversely, when packet evidence is correlated with additional data collected from network monitoring tools, firewall logs, and thorough data inspection analyzers, it yields robust forensic documentation.

Nonetheless, in case of a dispute, integrity (and authentication) of the digital evidences created for packets must be demonstrated. Some (old) schemes, such as Schneier & Kelsey [4], proposed solutions for keeping an audit log on an insecure

This work was supported by project SERICS (PE00000014) under the MUR National Recovery and Resilience Plan funded by the European Union - NextGenerationEU.

machine, so that log entries are protected even if the machine is compromised. Nowadays, modern techniques employing recent advances in cryptography and trusted computing could be exploited. For instance, to protect the cryptographic keys used for creating the evidences, secure cryptographic chips, like the Trusted Platform Module (TPM), could be leveraged. The timing information related to the captured packet flows must also be safeguarded against threats to the integrity of the timing data. [5]. On one hand, there are numerous applications that need to know the time a packet arrives, that is, when it was detected on a network link or at a device interface, while not having very stringent demands for time accuracy. Conversely, a specific group of applications (such as those in military, finance, or cyberattack investigations) have strict requirements for the authenticity and integrity of time and potentially the order of packets. Such applications may necessitate digital evidence that holds probative value for their internal network activities as well as for traffic coming from external sources.

This document builds upon earlier research [6] and outlines the initial design and execution of TC-NetTrack, which runs modules for capturing and filtering packets, generating and verifying secure timestamps related to packet flows, and producing cryptographically secured digital evidence for packets, referred to as the Digital Evidence (DE) Creator. We explore a method for the DE Creator to digitally sign flows, utilizing a technique previously suggested, known as tree chaining [7], alongside Merkle trees. To balance speed and space in packet signing, we employ optimized algorithms for setting up and traversing Merkle trees. We document the performance measurement of the solution implemented for the DE Creator, operating across a range of platforms and operating systems.

II. TC-NetTrack Design

The probative value of data hinges on several key attributes: authenticity, integrity, accuracy, completeness, compliance with the law, and access control. Authenticity refers to the ability to demonstrate that the data was recorded by a specific device. Integrity signifies that it can be shown that the data was captured without any alterations and has remained unchanged since then. Accuracy means the data is a faithful copy of the original, including the associated time information. Completeness indicates that all pertinent information (usually defined through a specific agreement) has been gathered, with nothing left out. Compliance with the law implies that the data has been collected and managed in accordance with all relevant legal regulations, including privacy laws. Lastly, access control ensures that the data is accessible only to those individuals who are authorized to view it. These properties must ultimately be upheld for all data (both input and output) processed by TC-NetTrack.

TC-NetTrack operates on two raw inputs: IP packets and time. Thus, the TC-NetTrack must check the abovementioned properties on the inputs, and exploit mechanisms to achieve them. For instance, for law compliance, some packets might have to be anonymized or filtered to remove personal information, while authenticity and integrity are typically achieved

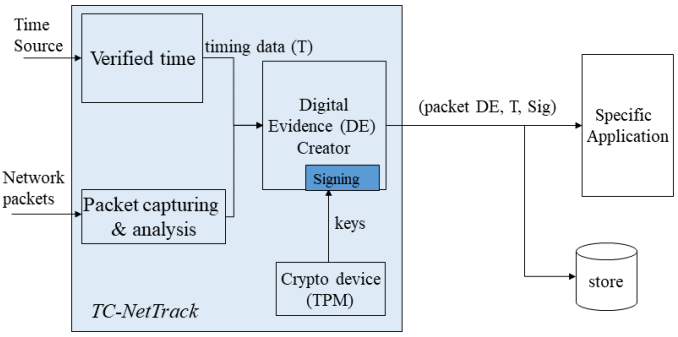

Fig. 1. TC-NetTrack components. The Digital Evidence (DE) Creator, and in particular the Signing submodule will be addressed in this work.

through cryptographic algorithms and keys. Regarding the output, the TC-NetTrack produces a digital evidence in the form *(packet DE, T, Sig)*, which is serving as a probative value that the captured IP packet was traced at time T on the TC-NetTrack's interface, and *Sig* is the cryptographic data for integrity and authenticity of each DE.

In a wider approach, the TC-NetTrack should be *fine-grained*, that is the device should operate on the smallest unit of data handled in the IP networks, that is, the IP packets belonging to packet flows [7]. Moreover, it should produce timestamped digital evidence, that is attest when an IP packet was "seen" on a link or network interface at a specific, precise time. In this regard, precise and validated timestamps should be linked to digital evidence to accurately represent the sequence of packets. The evidence must ensure the authenticity and integrity of the data. To avoid the creation of evidences with stolen or leaked keys in the creation of Sig, the cryptographic keys must be stored on a secure area of a device, like a TPM defined by the Trusted Computing Group. TPMs are cryptographic chips increasingly used to provide secure storage of sensitive data (keys) and support also basic cryptographic operations, like symmetric encryption, hash functions, and digital signatures. We assume the TPM as root of trust for the platform running the Digital Evidence (DE) Creator.

TC-NetTrack is composed of three main modules, as shown in Fig. 1: `Packet capturing & analysis`, `Verified time`, and `Digital Evidence Creator`. In the following, we will describe briefly these modules.

A. Packet capturing & analysis

.

Full packet capture is compulsory when inspecting what happened in a network (or a network node interface) in a particular time instant and to determine who was actually involved in a network activity. The interception, analysis, and backtracing of network packets are a considerable component of network forensics [8].

Network packets can be captured and analyzed using various packet capturing and analysis tools, which could be utilized

in the `Packet capturing & analysis` module of TC-NetTrack. There are both hardware devices and software solutions available for packet analysis, although software tools tend to be more adaptable and are widely used compared to hardware options that may be physical (dedicated or embedded), hybrid, or virtualized. Software tools for packet analysis include intrusion detection systems, network monitoring applications, network scanners, proxy servers, and tools for vulnerability assessment.

Some examples of packet analyzers software that could be exploited in our device are: *Wireshark*, one of the most widely used graphical packet capture and analysis tool, *Snort*[1], a free open source network intrusion detection/prevention system which can capture and process network traffic, *Suricata*[2], a highly-popular open-source tool for intrusion detection, *EtherApe*[3], a packet sniffer and network monitoring tool, for Unix, or *Tcpdump*[4], a widely-used software for capturing and dumping network packets for later analysis.

B. Timestamping IP packets

Alternative hardware measurement systems, like the DAG devices mentioned in [9], offer higher accuracy regarding time precision. To accurately represent the sequence of incoming packets at a network node interface, these devices capture and **timestamp** the network packets with as much precision as possible. The internal timing mechanism of a computer that relies on crystal oscillators is inadequate for providing accurate time readings due to the drift effect, where even slight temperature variations can alter the frequency by several hundred Hertz. Additionally, the skew in the clock's frequency can result in a timekeeping error exceeding 8 seconds within a 24-hour period, which is significantly inferior to that of a standard digital watch. In addition, the timestamps may be imprecise due to errors resulting from interrupt latency, buffering, and the processing times of the hosts utilizing software network measurement tools.

Endace measurement systems[5] produces specialized DAG cards that feature dedicated hardware, allowing them to capture packets at line rate with network speeds of up to 10 Gbit/s. Unlike standard network adapters, DAG cards can retrieve and map network packets to user space using a zero-copy interface, minimizing the need for interrupt processing.

IP packet timestamp resolution. Since the smallest event on a network is a single bit, the best possible resolution would be a clock operating at the bit-rate of the physical layer, allowing for a (unique) timestamp for each bit. However, most network protocols express messages in bytes, such as IP packets, which are defined as an integer number of bytes long. If we consider the smallest networking event to be a byte, a clock functioning at the byte rate of the medium would be adequate.

When using a computer architecture for monitoring networks and timestamping packets, the resolution of the timestamps assigned to the IP packets is dictated by the frequency of the system's clock [9]. For instance, on a 10Mb/s connection, the minimum size packet is transmitted in just 57.6 μs. This means that with a timestamp resolution of 10ms, numerous sequential packets may end up with identical timestamps. In such cases, the order of packet arrivals is not clear from the timestamp, leading to potential loss of ordering information during processing.

When multiple packets have the same timestamps, it becomes impossible to accurately generate distributions of packet inter-arrival times. The basic requirement is that the timestamps for two consecutive minimum-sized packets must differ. This ensures that the ordering of packet arrivals can be determined solely based on the timestamps of the packets. Consequently, the resolution of the timestamps must be less than the transmission duration of a minimum-sized packet over the medium.

Certain kernels of Linux (and FreeBSD) are capable of timestamping packets with a resolution of 1µs, which comfortably satisfies the minimum requirement for 10Mb/s Ethernet. For 100Mb/s Ethernet, the minimum packet transmission time is 5.7µs, so kernel clocks can also fulfill this requirement. Nevertheless, relying solely on the system clock for timestamping packets is not ideal, as the source of the time information lacks authentication or certification. The timestamp's time would simply reflect the internal clock of the PC monitoring the network, and this could potentially be exploited by malicious actors [10]. Therefore, it is essential to implement additional checks to ensure the authenticity and integrity of the timing information. To address the timestamp resolution issue, DAG cards may be utilized, as these devices feature a clock that offers timestamping resolution that is shorter than the transmission duration of the smallest packet, even on high-speed networks. Consequently, DAG cards facilitate complete line-rate data capture at speeds reaching up to 10 Gbit/s.

If we assume that TC-NetTrack is sampling a network at 100 Mb/s, and a packet of 1024 bytes, then approximately 12,000 packets/s are conveyed through the network. Let's assume that digital evidence should only be produced for a portion of the packets from a flow, specifically 10% of the packets (note that packet filtering and analysis are not discussed here). As a result, approximately 1,200 packets in a flow must be digitally signed within one second. We seek a solution that is both efficient and scalable for the quick digital signing of IP packets while also conserving space, as TC-NetTrack is expected to have constrained memory resources.

C. Digital Evidence Creator

The tree chaining technique proposed in [7], could be utilized by TC-NetTrack for signing and verifying multiple packets in a flow. This approach permits the signing and verification of a set (or *block*) of packets and possesses several features that make it appropriate for our purposes: i) packets in a flow can be verified individually, meaning a receiver does

[1]https://www.snort.org
[2]https://suricata-ids.org
[3]https://etherape.sourceforge.io
[4]https://www.tcpdump.org
[5]http://www.endace.com/

not need to receive all preceding packets in the sequence to validate a signature on a packet; ii) the size of packet signatures tends to be small, so the signatures included with each packet do not impose significant communication overhead; iii) both signing and verification processes are streamlined, making them suitable for real-time flows, such as those in TC-NetTrack; iv) each packet in a flow can be used immediately upon receipt, and it's possible for a receiver to obtain only a portion of all packets in that flow. Given that TC-NetTrack filters packet flows for delivery to various receivers, it is quite likely that different receivers will receive distinct subsequences from a flow.

The tree chaining method generates a block digest that is digitally signed using an asymmetric algorithm such as RSA or DSA, along with asymmetric keys. To ensure that packets can be verified individually, each packet must contain its own authentication details, which include the signed block digest (referred to as the *block signature*) along with some chaining data to demonstrate that the packet is part of the block. As the block signing and verification process is distributed across multiple packets, this approach significantly enhances the signing and verification speeds by several orders of magnitude compared to handling each packet independently. We have enhanced this approach by implementing a more efficient algorithm for constructing and traversing the hash tree, leading to a reduced *chaining time* for the signer. Specifically, we employed Szydlo's algorithm [11], which requires $log_2 N$ time and $3log_2 N$ space for each packet digest output.

Tree chaining technique details. By grouping *N* packets in a packet flow in a block, the block digest is computed as the root node of a Merkle hash tree. For example, for a block of eight packets p_1, ..., p_4, the packet digests H_1, ..., H_4, are assigned to the leaves of a Merkle tree of height two, while the internal nodes are computed as hashes of their children, as shown in Fig. 2. For example, the parent of the leaves H_1 and H_2 is $H_{1-2} = h(H_1|H_2)$ where $|$ denotes concatenation and h is the hash function used. The root of the tree is the block digest, while the *block signature* (BS) denotes the signed block digest.

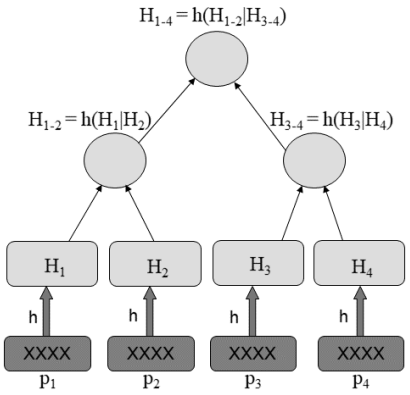

Fig. 2. Merkle hash tree with four packet digests stored in the tree leaves.

In order for a packet to be verifiable on an individual basis, it must contain its own *authentication data*, also referred to as the *packet signature* (PS). A packet signature includes the block signature, the packet's position within the block, and the sibling hashes of each node along the path from the packet to the root at each level. For example, the packet signature for p_1 includes the block signature (which is the digital signature applied to H_{1-4}), the position in the block (which is 1), and the PS containing H_2 and H_{3-4}. To authenticate the packets in a block, the verifier must construct a Merkle hash using the packet signatures. For instance, let's examine the process of verifying the third packet digest. The receiver calculates the digest H'_3 for the received packet and subsequently computes the digests of its ancestral packets in the tree. Specifically, H'_{3-4} is derived from h(H'_3,H_4), and H'_{1-4} is obtained from h(H_{1-2},H'_{3-4}), where H_4 and H_{1-2} are included in the packet signature. The verifier then invokes the verification function to check if H'_{1-4} matches the block digest H_{1-4} present in the block signature Sign(H_{1-4}), with Sign referring to the conventional digital signing operation, such as utilizing an asymmetric key pair.

The concept of *chaining overhead* refers to all the data in a packet signature apart from the block signature, which constitutes the PS for a single packet. It is important to observe that the quantity of elements in PS matches the height of the tree. Furthermore, the total number of hash operations needed to verify the accuracy of a packet digest (leaf value) also corresponds to the height of the tree.

The time required to construct an authentication tree for a set of *N* packets—covering the time needed to compute the packet digests—is referred to as *tree build time*. The block signature is generated by digitally signing the block digest located at the root, while each packet's signature is derived from the authentication tree and the block signature. The time taken to generate a packet signature is known as the *packet signature build time*, whereas the *chaining time* for a block at a signer consists of the sum of the tree build time and the packet signature build time for all packets within the block (not including the signing time of the block digest). For a verifier, the *chaining time* for a block is the total of the tree build time and the time needed to validate the chaining information in the packet signature for each packet in the block.

For TC-NetTrack, we require a solution that is efficient in both the time taken to construct and traverse the Merkle tree for a block of N packet digests and in the storage of the tree's nodes.

III. Digital Evidence Creator Implementation

R. Merkle proposed a method for signing multiple messages without the enormous cost of storing secret values per bit to be signed. His approach uses a binary tree, where each node is linked to a bit-string. Today, Merkle trees are widely implemented in various security frameworks, such as the Certificate Transparency system, for blockchain applications, and in key management systems. During an initial setup phase, numerous secret values are chosen and the results from a

hash function applied to each are made public. A supposed secret leaf value can be "validated" in relation to the root. While each node value is deemed public, a *Verifier* needs only to know the root value; a *Prover* provides the necessary public information to the *Verifier*: specifically, the values of all sibling nodes along the pathway to the root (known as the *authentication path* or *data*). Merkle trees are utilized in tree chaining technique, but traversing the tree should be conducted in a manner that optimizes both space and time. Traversing Merkle trees involves discovering an effective algorithm to generate the authentication data (at the signer) for consecutive leaves. Several algorithms (summarized below) have been proposed for this scope.

a) The classical algorithms: Classical algorithms, like those introduced by R. Merkle in [16] [17], show that producing the root node in a Merkle tree, often referred to as the *setup* phase, incurs a cost. Specifically, a tree with a height of H contains $N = 2^H$ leaves, and a general node at height h (where $0 \leq$ h $<$ H) relies on 2^h leaf values. Consequently, it is necessary to compute all N leaf values in addition to performing $2^H - 1$ hash function evaluations to derive the root value. One issue is the tacit assumption that all node values are retained, but it is clear that an effective Merkle tree contains numerous leaves, as seen in TC- NetTrack. Retaining all node values would take up significant space: for N leaves, the tree would require $N = 2^H$ leaves in addition to $N - 1$ internal nodes. Furthermore, the choice to rebuild the Merkle tree from scratch to generate the authentication data for each leaf demands considerable time; specifically, an interior node positioned close to the top of the authentication tree necessitates $2^H - 1$ hash operations. During the *traversal* phase, the signer produces a series of outputs, specifically the values of all sibling nodes along the route to the root, with one output generated for each leaf. Every output consists of two parts: the *leaf pre-image* and the *authentication path* associated with the leaf, which the receiver utilizes to validate the value of the leaf pre-image. The verifier must calculate the values of its ancestors by repeatedly applying a hashing process, and the leaf pre-image (i.e., the packet digest) is considered valid by the verifier if the obtained root value matches the provided/published digitally signed root value. We note that neither storing all node values (which could lead to an exponential increase in space taken depending on the height of the tree) nor recalculating the node values on-demand to determine the authentication path (which would also result in an exponentially increasing time cost) would be efficient for the Prover.

In terms of efficiency, creating an effective traversal algorithm on the *Prover* side is crucial:

- it should generate authentication data for each leaf sequentially (during round i, it outputs the value of leaf i along with its corresponding sibling nodes);
- the *Prover* may have limited memory capacity;
- the *Prover* should compute few *HASH* values per round.

Considering these aspects, certain algorithms employ amortization techniques: they utilize space and time-efficient methods for node calculations, where demanding nodes are processed across multiple iterations, with each iteration producing the output needed for one leaf verification. In the traditional traversal method, during each iteration, the algorithm provides the value of a leaf along with its corresponding sibling values, discarding the "expired" sibling values, and for each level, it focuses on preparing the "upcoming" siblings. With this approach, the upcoming sibling values must be ready in a timely manner.

To reduce the costs associated with traversing a Merkle tree, a function known as *TREEHASH* is utilized. First, *TREEHASH* derives the value of a leaf node n by employing the *LEAF-CALC* function, which provides the value of the leaf. Subsequently, *TREEHASH* processes the leaves in order, calculating the values of interior nodes whenever feasible and discarding nodes that are no longer required. Specifically, it saves the values of the nodes onto a stack and continuously applies the hash function *HASH* to sibling nodes that are at the same level. The traditional algorithm for traversing a Merkle tree executes one instance of *TREEHASH* for every height h ($0 \leq$ h $<$ H) to determine the next right node value. In each round, the *TREEHASH* state undergoes a modification through a single computational step. After every 2^{h+1} rounds, the calculation of the node value will be finalized, and a new instance of *TREEHASH* will initiate for the subsequent right node value. Using this classical algorithm, there may be as many as H instances of *TREEHASH* active simultaneously, corresponding to each height lower than H. Since the stack utilized by *TREEHASH* can include up to $h + 1$ node values, it is ensured only a space limit of $1 + 2 + 3 + ... + N$. For a tree of height H, *TREEHASH* demands a total of $2^H - 1$ computational units, taking into account that *LEAF-CALC* has a time expense equivalent to a hash function computation. The nodes in the authentication tree are eliminated when they are no longer necessary, thus *TREEHASH* requires only $H + 1$ hash values for storage.

The outcome of this method incurs estimated space and time costs of $O(log(N))$=$O(H)$ time for each round, where a tree has $N = 2^H$ leaves. Given that there is a round corresponding to each leaf, the algorithm's total time cost amounts to $O(Nlog(N))$, with the space used by nodes limited to $O(log(N)^2) = O(H^2)$. In these assessments, the definition of one unit of space corresponds to the space required to store the value of a node, while one unit of time represents the duration needed to compute a leaf's value via the function *LEAF-CALC*, or to derive the value of an interior node through the function *HASH*. For medium-sized trees, this algorithm maintains a fair balance of computational and storage efficiency; however, it is not considered "optimal". In fact, other methods with lower costs have been suggested, which are outlined briefly below. The difficulty with traversing Merkle trees lies in guaranteeing that all node values are prepared when necessary while being calculated in a way that conserves both space and time.

b) Logarithmic algorithms: Other works [12], [13], [14] enhanced the traversal of Merkle trees, enabling more efficient

management of large hash trees. These algorithms usually involve three main stages: *root generation*, *output* (which includes traversal), and *verification*. They provide improved balances between storage and computational needs during the traversal phase, which is responsible for producing the output that includes the *leaf pre-images* and the *authentication paths* in sequence. The fractal algorithm outlined in [12] necessitates a maximum worst-case computational effort of $T_{max} = 2logN/loglogN$ hash function evaluations for each output, along with a storage requirement of less than $Space_{max} \leq 1.5log^2N/loglogN$ hash values for a Merkle tree containing N leaves. In contrast, the recursive algorithm detailed in [13] operates in $O(logN/h)$ time and requires $O((logN/h)2^h)$ space per round, utilizing an arbitrary parameter h to divide the Merkle tree into L (where $L = H/h$) sub-trees of height h for their initial computation and storage. Therefore, this algorithm is recognized as having the highest complexity regarding both space and time.

M. Szydlo introduced an algorithm [11], known as Log-Log, which sequentially calculates the tree leaves and authentication path data, thus necessitating only $Log_2(N)$ basic operations for each round and the retention of $3Log_2(N)-2$ node values.

The concept behind this algorithm is to schedule concurrent *TREEHASH* calculations so that at any given round, the associated stacks are mostly empty. The schedule favors the computation of $Need_h$ (a node at level h) for lower h, but delays beginning of a new *TREEHASH* instance until all stacks $\{Stack_i\}$ are partially completed, containing no tail nodes of height less than h. Thus, the upcoming node values will be ready on time. The preserved node values are categorized based on the distinct functions they serve in the algorithm. The types of nodes include: *authentication*, *history*, *needed*, and *tail* nodes. First of all, for each height, $0 \leq i < H$, there is an authentication node, $Auth_i$, which is part of the authentication data to be output. Secondly, expired authentication nodes may be retained as a history node, $Keep_i$, for the purpose of speeding up a left node computation. A completed upcoming right node is denoted $Need_i$. and there may be some intermediate tail node values stored in $Stack_i$, the stack associated to an incomplete *TREEHASH* computation. This algorithm consists on $N = 2^H$rounds, one for each leaf, and the index *leaf* is used to denote the round number. The index *active* is used to focus on a particular height i, because multiple nodes are being calculated concurrently.

This traversal algorithm consists also of three phases: key generation, output, and verification. During *key generation*, the root of the tree and the first authentication path are computed. The *output* phase consists of N rounds, one for each leaf. During round *leaf*, the leaf's value, $\Phi(leaf)$ (or leaf pre-image) is given as output, as well as the authentication path, $\{Auth_i\}$. The *verification* phase is identical to the traditional verification in the classical algorithms.

IV. Experiments

We selected the Log-Log algorithm since other methods [11] [12], [13] cannot achieve both $time = O(Log(N))$ and $space = O(Log(N))$ simultaneously. An alternative algorithm is [14], which purports to provide the optimal balance regarding time and space complexity, or the sparse Merkle trees [18], or Merkle^2 [15], that we planned to assess in future research.

We adopted a layered approach to implement the *Log-Log* algorithm. We defined and utilized a `node` structure that includes a hash value representing a node in the tree (for example, the packet digest for a leaf) and the node's index within the tree. Following this, we executed the commonly used tree-building method known as the TREEHASH algorithm, introduced by Merkle himself. The TREEHASH employs a stack, and we subsequently developed a `nodestack` based on the `node` structure. Lastly, we created a module called `merkle` that incorporates functions for managing Merkle trees (such as initialization, setup, output, and verification) and is easily accessible for use by external applications.

Szydlo used hash size and computation time as units of space and time complexity, and provided a formal proof that time cost is bounded by log_2N and space cost is bounded by $3log_2N$. By applying his assumption to our implementation, we can state that the time cost increases each time the `node_hash` function, responsible for generating a hash, is executed; likewise, the space used grows with each call to `node_create` and declines with each invocation of `node_destroy`. By utilizing the function from the `logger` module that evaluates both space and time costs in terms of their respective units, we derived an accurate assessment of the total time and maximum space costs represented in these units. These assessments are independent of the specific machine or operating system and remain unaffected by unpredictable factors such as other applications running concurrently.

The time obtained for traversing a Merkle tree with $N = 2^{20}$ leaves, which aligns with about 1 million packet digests, across various platforms, is presented in Table I. In the worst-case scenario, utilizing the Sun 4U Enterprise 250 platform, establishing a Merkle tree with 131,072 leaves required 2.55 seconds, while setting up a Merkle tree with $N = 2^{20}$, or 1,048,576 leaves, took 20.62 seconds. When it came to traversing the Merkle tree on the same architecture, the time taken was 24.36 seconds for a tree containing 131,072 leaves and 243.45 seconds for a tree with 2^{20} leaves.

In the worst-case scenario on the Sun 4U Enterprise 250 platform, building a Merkle tree for 131,072 packets took roughly 2.5 seconds (unlike [7], this duration includes the time needed to compute the packet digest values for the leaves). The time required for signing the block digest was 0.0655 seconds when using 1024-bit RSA and 0.0345 seconds with 1024-bit DSA. These key sizes should be sufficient for the short term, as we expect the packet signatures to be verified shortly after they are generated. The timing for signing the block digest was determined using the "openssl speed" utility tool on the aforementioned platform. Therefore, the chaining time at the signer amounts to about 26.92 seconds (calculated as 2.5 + 0.06 + 24.36) for a block containing 131,072 packets. As a result, the average time taken to sign a packet is 26.92/131,072 = 0.20 ms. In terms of latency, the

Model / CPU Speed / RAM	Time [seconds]
AMD Turion / 1.8 GHz / 1 GB	31.07
Intel Core2 / 2.4 GHz / 2 GB	41.39
Sun4U Enterprise 250 / 2 X UltraSPARC-II 400 Mhz / 512 MB	243.45
Dual Intel Pentium 4 / 3GHz / 1 GB	22.26

TABLE I
TIME FOR TRAVERSING A MERKLE TREE WITH 2^{20} LEAVES.

TC-NetTrack would generate digital evidence for the 131,072 packets within 30 seconds at the worst-case scenario, whereas the delay for other platforms would be considerably lower. For instance, on a Windows XP system with an Intel Pentium Mobile 1.7 GHz processor and 1 GB of RAM, the time taken to chain the 131,072 packet digests at the signer totaled 1.076 + 0.004 + 8.360 = 9.44 seconds, resulting in an average of 0.07 ms per packet digest for signing. Wong et al. [7] points out that for real-time applications, the duration T within which no more than N packets can be produced must exceed T_{sign} + $chain_s(N)$ at the signer, where T_{sign} represents the time required for signing the block digest and $chain_s(N)$ indicates the chaining time for a block comprising N packets. This condition can be satisfied by our implementation on mid-range to high-performance platforms.

To verify the packets in a block, a receiver must construct a Merkle tree using the packets and validate their signatures. The term *chaining time* refers to the duration needed for the receiver to validate the chaining information found in the packet signature of every packet within the block. For a block containing 131,072 packets, the measured *chaining time* at the verifier on the slowest platform examined (Sun4U) was 15.49 seconds dedicated to verifying the chaining information for each packet digest in the block. The time taken to verify a signature (as measured by the "openssl speed" utility) is 0.0036 seconds for the 1024-bit RSA algorithm and 0.0434 seconds for the 1024-bit DSA algorithm. Therefore, the total time required to verify a block containing 131,072 packet digests was 15.498 + 0.003 = 15.501 seconds. As a result, the average time to verify one packet is 0.118 milliseconds, which is significantly quicker compared to the 3.6 milliseconds needed for individual signature verification using the 1024-bit RSA algorithm for each packet. On a Windows XP platform with a 1.7 GHz CPU and 1 GB of RAM, the total chaining time (at the verifier) for the 131,072 packet digests was 6.365 + 0.0002 = 6.3652 seconds, averaging 0.04 milliseconds per packet.

V. CONCLUSIONS AND FUTURE WORK

This paper proposes TC-NetTrack, a trusted computing-enabled device for creating digital evidences on network packets. Ensuring the authenticity and integrity of the network traffic recorded by TC-NetTrack is difficult, as signing each individual packet can lead to significant delays, while signing groups of packets imposes additional and considerable storage demands. We have implemented a method that leverages Merkle trees for the digital signing of packet flows. Furthermore, we have enhanced its implementation by using an optimized Merkle tree traversal algorithm, which allows for fast creation of 'proofs' for the signed packet flows while saving space. Future work will cover the exploitation of TPM keys and crypto functions in TC-NetTrack and integration of secure time to protect the integrity of timing data associated with the digital evidences.

REFERENCES

[1] A. Orebaugh, G. Ramirez, J. Burke, L. Pesce, J. Wright, and G. Morris, "Wireshark & Ethereal Network Protocol Analyzer Toolkit," Syngress, Rockland, MA, USA.https://www.oreilly.com/library/view/wireshark-ethereal/9781597490733/

[2] L. F. Sikos, "Packet analysis for network forensics: A comprehensive survey," Forensic Science International: Digital Investigation, Volume 32, 2020, 200892, ISSN 2666-2817, doi: 10.1016/j.fsidi.2019.200892.

[3] H. Kim, E. Kim, S. Kang, and H. K. Kim, "Network Forensic Evidence Generation and Verification Scheme (NFEGVS)," Telecommun. Syst. 60, 2 (October 2015), 261–273. https://doi.org/10.1007/s11235-015-0028-3.

[4] B. Schneier, J. Kelsey, "Secure audit logs to support computer forensics," ACM Transactions on Information and System Security (TISSEC), Vol. 2 , Issue 2, May 1999, pp: 159-176, doi: 10.1145/317087.317089.

[5] D.G. Berbecaru and A. Lioy, "Attack Strategies and Countermeasures in Transport-Based Time Synchronization Solutions.," In: Camacho, D., Rosaci, D., Sarné, G.M.L., Versaci, M. (eds) Intelligent Distributed Computing XIV. IDC 2021. Studies in Computational Intelligence, vol 1026, 2022, Springer, Cham, doi: 10.1007/978-3-030-96627-0_19.

[6] D. Berbecaru, "On creating digital evidence in IP networks with NetTrack," in Handbook of Research on Network Forensics and Analysis Techniques, IGI Global, pp. 225 - 246, doi: 10.4018/978-1-5225-4100-4.ch012.

[7] C.K., Wong and S.S. Lam, "Digital Signatures for Flows and Multicasts," IEEE/ACM Transactions on Networking, Vol. 7, No.4, August 1999, pp: 502-513.

[8] B.J. Nikkel, "Generalizing sources of live network evidence," Digital Investigation, Volume 2, Issue 3, 2005, pp: 193-200, doi: 10.1016/j.diin.2005.08.001.

[9] S.F. Donnelly, "High Precison Timing in Passive Measurements of Data Networks," Ph.d. Thesis, University of Waikato, Hamilton, New Zeeland, 2002.

[10] D.G. Berbecaru et al., "Mitigating Software Integrity Attacks With Trusted Computing in a Time Distribution Network," in *IEEE Access*, vol. 11, pp. 50510-50527, 2023, doi: 10.1109/ACCESS.2023.3276476.

[11] M. Szydlo, "Merkle Tree Traversal in Log Space and Time," in Proc. of Eurocrypt 2004, LNCS vol. 3027, pp:541-554.

[12] M. Jakobsson, T. Leighton, S Micali, and M. Szydlo, "Fractal Merkle tree Representation and Traversal," in RSA Cryptographers Track, RSA Security Conference 2003, Available at: https://link.springer.com/content/pdf/10.1007/3-540-36563-X_21.pdf

[13] P. Berman, M. Karpinski, Y. Nekrich, "Optimal Trade-Off for Merkle Tree Traversal," in Report No. 49, Electronic Colloquium on Computational Complexity 2004, doi: 10.1007/978-3-540-75993-5_13.

[14] P. Berman, M. Karpinski, Y. Nekrich, "Optimal trade-off for Merkle tree traversal," Theoretical Computer Science, Elsevier Science Publishers, Vol. 372 , Issue 1, March 2007, pp: 26-36, doi: 10.1007/978-3-540-75993-5_13.

[15] Y. Hu, K. Hooshmand, H. Kalidhindi, S. J. Yang and R. A. Popa, "Merkle2: A Low-Latency Transparency Log System," 2021 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 2021, pp. 285-303, doi: 10.1109/SP40001.2021.00088.

[16] R. Merkle, "Secrecy, Authentication and Public Key Systems," UMI Research Press, 1982. Also appears as a Stratford Ph.D. thesis in 1979.

[17] R. Merkle, "A digital signature based on a conventional encryption function," in Proc. of A Conference on the Theory and Applications of Cryptographic Techniques on Advances in Cryptology, LNCS Vol. 293, 1987, pp: 369-378, doi: 10.1007/3-540-48184-2_32.

[18] R. Dahlberg, T. Pulls, R. Peeters, B. Brumley, and J. Roning, "Efficient Sparse Merkle Trees: Caching Strategies and Secure (Non-)Membership Proofs," in Proc. of the 21st Nordic Workshop on Secure Computer Systems (NORDSEC 2016) (Vol. 10014, pp. 199–215). Springer Verlag. doi: 10.1007/978-3-319-47560-8_13.