Applying distributed ledger technology to digital evidence integrity

William Thomas Weilbach and Yusuf Moosa Motara

Abstract—**This paper examines the way in which blockchain technology can be used to improve the verification of integrity of evidence in digital forensics. Some background into digital forensic practices and blockchain technology are discussed to provide necessary context. A particular scalable method of verifying point-in-time existence of a piece of digital evidence, using the OpenTimestamps (OTS) service, is described, and tests are carried out to independently validate the claims made by the service. The results demonstrate that the OTS service is highly reliable with a zero false positive and false negative error rate for timestamp attestations, but that it is not suitable for time-sensitive timestamping due to the variance of the accuracy of timestamps induced by block confirmation times in the Bitcoin blockchain.**

Index Terms—**Digital forensics, blockchain, evidence integrity.**

I. INTRODUCTION

IN the face of an impending financial crisis, an anonymous researcher, going by the pseudonym Satoshi Nakamoto, proposed a cryptographic solution to the problem of distributed trust, also known as "The Byzantine Generals Problem" [1]. This solution, in the form of blockchain technology, was presented in a paper titled: "Bitcoin: A Peer-to-Peer Electronic Cash System" [2]. Blockchain technology has emerged as a significant and potentially revolutionary technology inspiring a new class of solutions to problems all but forgotten.

The potential applications of blockchain technology are vast and continue to diversify every day with the emergence of smart contract platforms such as Ethereum [3] and digital currencies such as Zcash [4]. However, despite its widespread adoption, blockchain technology remains relatively unexplored in areas that extend beyond payments and currency. Blockchains solve a few fundamental issues of trust by operationally incorporating the properties of immutability and transparency, and when applied to other problem domains, these exact properties are equally valuable. One of those problem domains is the domain of digital forensics: "the discipline that combines elements of law and computer science to collect and analyse data from computer systems, networks, wireless communications, and storage devices in a way that is admissible as evidence in a court of law" [5].

This paper examines the application of blockchain technology to the field of digital forensics. More specifically, it identifies a particular requirement – proof of existence – within the field, and independently assesses its application and relevance in the context of the OpenTimestamps (OTS) system which could meet this requirement. Given the criticality of digital forensics in the broader space of cyber crime, is essential that software such as OTS be tested to provide assurances as to its reliability and accuracy since it must be assumed that these aspects of digital forensics technology will at some point be called into question as part of an investigation. It is therefore necessary to provide a conclusive and vetted explanation of the proof mechanism to pass peer review.

The paper is structured as follows. Section II describes the field of digital forensics and argues strongly for the importance of trustworthy and independently verifiable digital evidence. Section III then describes blockchain technology and the properties thereof. Section IV considers the proof-of-existence problem and is followed by a section that specifically focuses on OpenTimestamps. Independent testing of this software follows, and the paper concludes with some discussion and conclusions.

II. DIGITAL FORENSICS

The digital forensic process, can, at a high level, be described by three basic practices: acquisition, analysis, and presentation [6]. The act of acquiring evidence is the first step in any digital forensic investigation and can be a non-trivial task at the best of times [7]. The acquisition phase is also arguably the most critical in any investigation, as any error here will naturally propagate to the following phases and potentially affect the integrity and admissibility of the evidence as a whole: any issue that adversely affects the admissibility of digital evidence can cast doubt on entire investigations [8].

A common, and sometimes mandated, practice during the acquisition phase is the act of hashing evidence [7]. A cryptographic hash, also referred to as a digest, is a unique, fixed-length value, generated from any evidentiary artefact of variable length (the pre-image), that can serve to identify that piece of evidence. A cryptographic hash is the product of a one-way deterministic mathematical function through which data of arbitrary length can be passed to produce a collision-resistant fixed length representation of that data [9]. A key property of a hash function is that a minor change in the input will result in a significant change in the fixed length output [10]. A second key property is that cryptographic hashes are computationally infeasible to reverse to determine a pre-image of a given hash [11]. Hashes are most commonly used to determine if the evidence has been tampered with between the time the hash was generated and when the evidence is scrutinised.

The presentation phase of the digital forensic process involves sharing or presenting the results to a selected audience, and includes showcasing and explaining the information concluded from the previous phases. The presentation phase of an investigation can be, and most likely will be, subjected to intense scrutiny regarding the integrity of evidence [12]. This is especially relevant if the investigation forms part of a criminal case. It is, therefore, of paramount importance that any observations presented be irrefutably backed up by facts

derived from evidence of which the integrity can be proved without a doubt.

III. BLOCKCHAIN

Satoshi Nakamoto proposed the Bitcoin blockchain upon which all subsequent blockchain implementations to date are based [2]. Fig. 1 is a simplified visual representation of a blockchain-type system.

In Fig. 1, there is no starting, or genesis, block, but rather a sequence of blocks at some point after the genesis block. It can be seen that one input into a block is the hash of the previous block. To further improve security, this hash is combined with a nonce and some arbitrary data items before it is once again hashed and provided as input to the following block. A nonce is simply a value used once for a particular message or operation [13], and is usually a random value [14]. By chaining blocks together like this, it is possible to verify the data in them, as any change in the data will result in a change of the hash which will necessarily cascade up the chain, changing all subsequent block hash values.

To explain general blockchain functionality further, the first implementation of a blockchain-driven system – Bitcoin – will be used. Although not all blockchains follow this exact model, they are all based on the same basic principles.

Blocks are collections of structured data that form a fundamental part of the ledger. A "miner" within the system can "mine" a block – thus obtaining a new block to append to the chain – by solving a computationally difficult puzzle that is associated with the latest block in the chain.

The *chain* is a series of connected blocks. Each block in the chain contains a collection of transactions, each of which contains a series of inputs and outputs. Fig. 2 is a high level view of blocks in the Bitcoin blockchain.

Transactions involve the creation or transfer of value within the network. Nodes that process transactions in the Bitcoin network are referred to as *miners*, and their function is to:

1) collect transactions that are broadcast to the network;
2) add those transactions to the block structure (see Table I);
3) solve a Proof-of-Work (PoW) puzzle associated with that block.

Bitcoin, like other forms of currency backed by commodities and resources, can suffer the effects of inflation should it be overproduced. Since Bitcoin is completely digital, there needs to be a mechanism to regulate the amount of Bitcoin released into the system. If Bitcoins were trivial to create, it would have little to no value as a store of value, since any person could simply create vast amounts of the currency. To combat the effects of inflation, Bitcoin is designed to be difficult to create through controlled supply, which is enforced in two ways: by having a finite supply of Bitcoin, and regulating the rate at which new Bitcoins can be mined. The Bitcoin generation algorithm defines at what rate currency can be created and any currency generated by violating these rules will be rejected by the network.

The Proof-of-Work is another important component of controlled supply as it ensures that the difficulty of finding a block can be adjusted to compensate for fluctuations in the network's aggregate mining power. By adjusting the difficulty every 2016 blocks – through consensus by all participating

miners – the network can respond to fluctuations in mining power and ensure that blocks are released, on average, every 10 minutes. The PoW puzzle implemented by Nakamoto [2] was based on the Hashcash system developed by Back [15]. As the mining power of the network increases, the difficulty of the PoW puzzle is adjusted to slow the rate of block creation. The PoW puzzle difficulty is defined by the *nBits* field (see Table I) of a block.

To solve a PoW puzzle, a miner must calculate a double-SHA256 (henceforth abbreviated as "dSHA256") hash H over the contents of the block b so that it is smaller or equal to the target value *nBits*. Since hash functions are deterministic, $\mathrm{dSHA256}(b)$ will always result in the same output. In order for a miner to generate different hashes to satisfy the condition, it must incorporate a random nonce n into the input of the function such that $\mathrm{dSHA256}(b \parallel n) \leq nBits$. Once the condition is met, the puzzle has been solved, and the miner broadcasts this block, accompanied by the proof, to the network for confirmation.

PoW difficulty is adjusted by decreasing the *nBits* value. The more zeroes required, the more hashing operations the miner has to perform in order to find a value that satisfies the condition. This is because there is no way, other than brute-force, to find a solution with the appropriate number of leading zeroes [11]. Thus, by solving this PoW, the miner proves that it has invested an approximate amount of effort at its own cost toward finding the block, and that it is a willing and conforming participant in the network.

This ongoing work results in the chain as depicted in Fig. 2. Due to the distributed nature of the system where many nodes compete to solve the PoW puzzle, it occasionally happens that more than one miner solves the PoW for different blocks at the same time. When this happens, it results in a *fork* in the chain; and each node will then accept the first proof it receives as the correct one and build the chain from that block. When this happens, the rejected block is called an orphaned block, depicted in Fig. 2 as the block with dotted borders. However, transactions that were part of these orphaned blocks are not lost but are instead rebroadcast to the network for later inclusion. Miners always work on the longest chain, which implies the chain on which the most computational effort was exerted. This is to ensure that there is consensus around which chain is the correct chain, and to prevent malicious nodes from altering previous blocks to create an alternative chain.

The exact structure of a block can be seen in Table I. A Bitcoin block can be up 1024 kilobytes (1024000 bytes) in size, but no larger. Blocks larger than 1024 kilobytes, are considered invalid and will not be accepted by the network. As can be seen from the size allocations in Table I, the header data for a block (all non-transaction data) can be up to 80 bytes in size, leaving the vast majority (1023920 bytes) for transaction data. The block structure has a direct effect on the way in which miners are incentivized to include a transaction, as well as being important for understanding the way in which OpenTimestamps reduces its costs.

Transactions are listed in the *vtx[]* part of the block, and described more fully in Table II. A transaction must be either a coinbase transaction or a transfer of value. Each transaction output is associated with a particular *scriptPubkey* which specifies, using a primitive and minimal language, the conditions under which that output can be spent. Usual conditions include
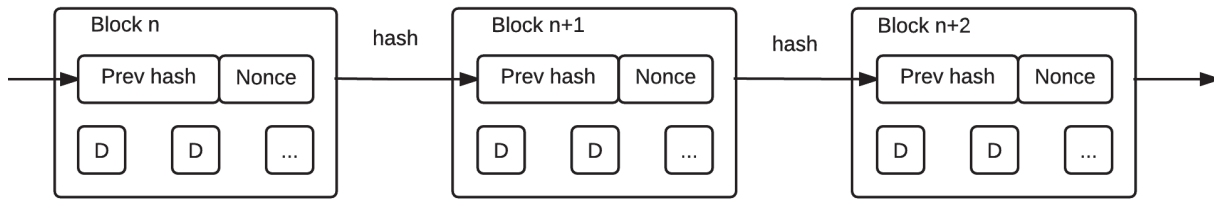
Fig. 1. A visual illustration of a blockchain

TABLE I
BITCOIN BLOCK STRUCTURE

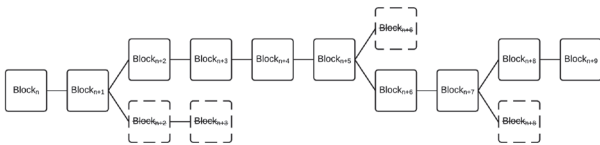|	Field Name	Type	Size	Description
Header	nVersion	int	*4 bytes*	The block format version
	HashPrevBlock	uint256	*32 bytes*	Hash of previous block header
	HashMerkleRoot	uint256	*32 bytes*	MR of all transactions
	nTime	unsigned int	*4 bytes*	UNIX-format time stamp of block creation time
	nBits	unsigned int	*4 bytes*	Proof of work problem target
	nNonce	unsigned int	*4 bytes*	Nonce for solving proof of work problem
Payload	cnt_vtx	VarInt	*1 to 9 bytes*	Transaction count in vtx[ ]
	vtx[ ]	Transaction	*Variable Array*	Array of transactions

Fig. 2. Example of a Bitcoin blockchain with forks

a way to authenticate the owner of the funds using public-key cryptography; however, any other scriptable conditions can be used. The *scriptSig* part of an input must fulfill the necessary conditions for an output to be spent. If an executed script does not return a *true* value when executed by a node in the mining network, then the transaction is invalid and is ignored. Both the *scriptPubkey* and *scriptSig* are under the control of the user and can, compared to other fields, store relatively large amounts of data [16].

There are economic incentives that explain why miners choose to mine blocks; there is more to be said about the exact structure of a block; there is more complexity to be understood around the increasing difficulty of PoW puzzles; and there is more depth to be examined relating to the mechanics of block verification which is covered in depth by [16]. However, these topics are not directly relevant to this work and have been elided. Instead, from a digital forensics perspective, what has been said is sufficient to understand four key properties of a blockchain: immutability, chronology, redundancy, and transparency.

Immutability, the lack of ability to be changed, is arguably one of the most important properties of blockchain systems. Immutability is not a property on the macro level - as the chain is constantly changing and expanding when new blocks are added - but rather on a more granular level as data and transactions that are embedded in the blocks are unchangeable. This immutability is conditional and strengthens over time as a consequence of the design of the system [9]. As newer blocks form on top of older blocks, the block depth increases and the ability to change data embedded in that block diminishes. Any entity that wishes to change some data within a block would have to change the data in that block and recompute that block and all subsequent blocks faster than all the other nodes in the network can. It would therefore be theoretically possible for multiple nodes to collude to change some data, but this type of collusion is unlikely and inherently detectable. In the Bitcoin blockchain, the current block depth to guarantee a permanent and unchangeable transaction is six-blocks deep [9]. This immutability means that the public ledger record cannot be altered to reflect a record that represents a false or fabricated transaction, and can thus be trusted. The immutability of the information embedded in the blockchain means that, to any observer or participant, all information can be considered an unchangeable and a true record of data over time.

Chronology – the sequential arrangement of events or transactions over time – is another property of blockchain design that gives it immense value and utility. Timestamping is the ability to associate the existence of a certain piece of information with a specific moment in time [17], and all blocks contain timestamps [2]. A mining node can reject a timestamp that is deemed to be too old or in the future, and timestamps are thus validated by distributed consensus.

TABLE II
BITCOIN TRANSACTION STRUCTURE

Field Name		Type	Size	Description
nVersion		int	*4 bytes*	Format version of the transaction
cnt_vin		varInt	*1-9 bytes*	Count of the entries in vin[ ]
vin[ ]	hash	uInt256	*32 bytes*	Hash of past transaction (dSHA256)
	n	uInt	*4 bytes*	Transaction index in the output of previous transaction
	ScriptSigLen	varInt	*1-9 bytes*	Length of ScriptSig
	scriptSig	Script	*Variable length*	Script that specifies conditions for the spending of the output
	nSequence	uInt	*4 bytes*	Sequence number of transaction
cnt_vout		varInt	*1-9 bytes*	Count of the entries in vout[ ]
vout[ ]	nValue	int64	*8 bytes*	Amount
	scriptPubkeyLen	varInt	*1-9 bytes*	Length of ScriptPubkey
	scriptPubkey	Script	*Variable length*	Script that specifies conditions for the output to be claimed.
nLockTime		uInt	*4 bytes*	Timestamp indicating the time past which transactions can be included in a block

By combining immutability in the form of an append-only chain and chronology in the form of trusted timestamping, blockchains give the unique ability to store and verify the existence of data at a point in time with accuracy.

Redundancy is a further significant property of a blockchain-based system, and was a key design consideration in the Bitcoin blockchain [2]. Not only would the system need to be fault-tolerant to be widely used, but it would also necessitate the participation of many entities to safeguard the decentralisation that lies at the core principle of the concept: trust. Decentralised trust, or the lack of trust in a single entity, implies that trust is the responsibility all participants and not that of a governing entity or a subset of privileged entities. Most blockchain-based systems therefore have incentive systems and each differ slightly in terms of reward. By having a completely distributed system with decentralised trust, the resiliency of the system can be guaranteed for as long as there is an incentive to participate in the system.

Transparency is the final of the four core blockchain properties and is more of a functional requirement and not a design consequence. All transactions need to be broadcast openly to any entity willing to listen. Furthermore, the information embedded inside the ledger must be open for all to see and verify. This is necessary for the Bitcoin system to work as a distributed financial ledger since transactions are stored instead of balances. Therefore, to calculate the balance of a specific address, all the transactions to and from that address need to be visible.

By combining immutability, chronology, redundancy, and transparency, blockchain-based systems are uniquely equipped to address many of the problems associated with trust and decentralised processing.

IV. PROOF OF EXISTENCE OF DIGITAL EVIDENCE

It is useful, at this point, to summarise the requirements of the digital forensics community and tie those to properties of the blockchain.

- **Existence**. It is important to verify the existence of digital evidence. The blockchain allows arbitrary data, including digital evidence, to be embedded within it. The transparency and immutability of the blockchain can ensure that the evidence is preserved for as long as the blockchain itself exists, and that the evidence can be examined by any party at any time.
- **Chronology**. It is important to verify that digital evidence existed at a particular point in time. The chronology of a blockchain, and the digital consensus around timestamps, can be used to show this.
- **Non-repudiation**. It is important that the digital forensics analyst cannot change a claim that is made. If this were the case, then the trust that is placed in digital evidence would rest solely on the reputation of the digital forensics analyst. The blockchain's immutability, transparency and redundancy makes it easy to make a claim that cannot be repudiated at a later date. This, in turn, ensures that the claim made before the analysis stage begins cannot be changed at the analyst's discretion.

The existence requirement is complicated by two issues: firstly, the evidence may sometimes be of a private nature, and may therefore not be revealed to the public; and, secondly, the evidence may be very large. The second issue ties in directly with the already-mentioned fact that a larger block payload is correspondingly more expensive to store in a blockchain. Both of these issues are addressed by a blockchain timestamping services such as Chainpoint [18], proof-of-existence (PoE) [19] and OpenTimestamps [20].

PoE or blockchain timestamping services embed the hash of arbitrary data – and not the data itself – into a block. By using this method, it is possible to permanently embed a small amount of data into the Bitcoin blockchain; the embedded data may also be prepended with some marker bytes that makes searching for such proofs in the blockchain easier. Of the various PoE services, OpenTimestamps (OTS) is the only one which is completely open source and transparent, and therefore

the only one which is open to public examination and testing – both of which are very important for a service of this nature in the context of digital forensics.

V. OpenTimestamps

The OTS service consists of server-side and client-side components that interact, using an open protocol, to perform the timestamping of data as well as validate existing timestamps for which proofs have been received. The client-side component takes some arbitrary data as input, hashes it, incorporates that hash into a predefined structure and submits it to the server-side component via remote procedure call (RPC). The server-side component then takes the data and incorporates it into a Bitcoin transaction and submits that transaction to be processed into the Bitcoin blockchain. The server then sends a OTS proof back to the client and the client can, from that point onward, use that proof to verify the timestamp and the integrity of the data by performing another RPC call.

In the OTS system, the Bitcoin blockchain acts as notary as it affords users thereof the ability to create and verify both the integrity of a document and the approximate date at which it must have existed. OTS allows any participant to submit the hash of an arbitrary piece of data to be embedded in a transaction in the Bitcoin blockchain and to timestamp that document hash on the blockchain. The accuracy of such a time stamp is estimated by to be within two to three hours of the submission date and time [20].

OTS uses "commitment operations" [20] which simply is a function that alters the function input to produce a deterministic output. A simple concatenation function such as $a||b = ab$ is an example of a commitment operation. In OTS, the verification of an OTS timestamp is the execution of the sequence of commitment operations and the comparison of the output to the value stored on the Bitcoin blockchain. OTS timestamps can therefore be said to be trees of operations with the root being the message, the edges (also known as nodes) being the commitments, and leaves being the attestations. The usage of these terms is not coincidence but rather as a result of the heavy reliance on Merkle Hash Trees (MHTs) [21] to support OTS functionality.

A MHT is a data structure that relies heavily on cryptographic hashing for its function and value. The broad purpose of a MHT is to make the validation of data more efficient, by providing a way for large amounts of data to be validated against a single hash value without having to rehash all the data. It is often used in peer-to-peer protocols to facilitate the validation of data without having to transfer vast amounts of data between peers on a bandwidth-restricted network. In this sense, the purpose of a MHT is to provide a mechanism for validating large sets of data in a distributed environment with reduced capacity for data storage, transfer and computation. Its application in blockchain technology is for this exact same purpose; and, in fact, it is used by the Bitcoin blockchain itself, as well as by the OTS application that is built upon the blockchain.

MHT consist of three basic components:

1) The root, also called the Merkle Root (MR), of which there is only one per tree

2) The nodes, also referred to as Child Nodes (H), of which there must be at least two; theoretical there is no maximum number of Child Nodes per tree

3) The leaves (L) of which there must be at least two; theoretical there is no maximum number of leaves per tree

Fig. 3 shows a basic example of a MHT with four leaves, six nodes and a root. For the purpose of explanation, the four leaves would be the raw data needing to be verified. This data is not included in the tree but serves as the basis of its creation. Theoretically, there can be an infinite number of leaves, but the number of leaves is usually limited to avoid long running computation. One level up (level MR-2) are the nodes, H_1 to H_4, which are hashes of the respective leaves (L_1 to L_4). It is essential to note that these nodes are hashes (one-way functions) of the leaves but that the actual hash algorithm is not stipulated. Each use case may call for different hash algorithms, based on the preference for speed over security, or vice versa. In the Bitcoin implementation and other implementations where security of the hash values (their resistance to collision) is important, hash algorithms, like SHA256, are used. One level up (MR-1) are the secondary nodes, which each consists of the hash of the concatenation ($H_{xy} = H_x||H_y$) of its children on MR-2. Finally, on the very top level is the MR which, like the nodes below it, is a hash of its concatenated children. It is considered the root as it is a single hash that incorporates elements of all the leaves. In this way, a seemingly insignificant change in a single leaf will propagate up the tree and result in a changed MR. It is clear that MR can be used to verify the integrity of all of the leaves independently or as a whole; therein lies the power of MHT as a mechanism for verification.

By using MHTs, a large amount of arbitrary data can be hashed into a single MR hash. To verify any leaf on the tree, its original data, the hashes on its path, and the root hash needs to be known. This means that not all the leaves need to be present to be able to validate the integrity of a single leaf, thereby allowing the MHT to preserve space.

OTS primarily makes use of MHTs to address the problem of scalability. By using MHTs, OTS can compress large amounts of data into a single hash by adding individual hashes as leaves of a MHT. These leaves would then be collapsed into the MHT root which, in turn, is embedded into a Bitcoin transaction. This aggregation occurs on OTS aggregation servers when the OTS client sends the hash of the desired data to at least two OTS aggregation servers. These aggregation servers collect all of the different hashes from different OTS clients, uses them as leaves of a MHT and computes the MR. This MR is in turn embedded into a single Bitcoin transaction.

Once a MR for a given set of leaves has been embedded in the Bitcoin blockchain, verifying any single leaf can be accomplished by simply replaying a subset of commitment operations with efficiency $O(log_2(n))$. Fig. 4 serves as a visual example of a series of relevant commitments to be able to prove the integrity and existence of data in L_2.

Note how, to verify the integrity or the timestamp associated with the data in L_2, only a subset of leaves or nodes need to be known. This means that many hashes representing large datasets can be stored within the bounds of a small amount of blockchain data by aggregating these leaves into a MHT. The
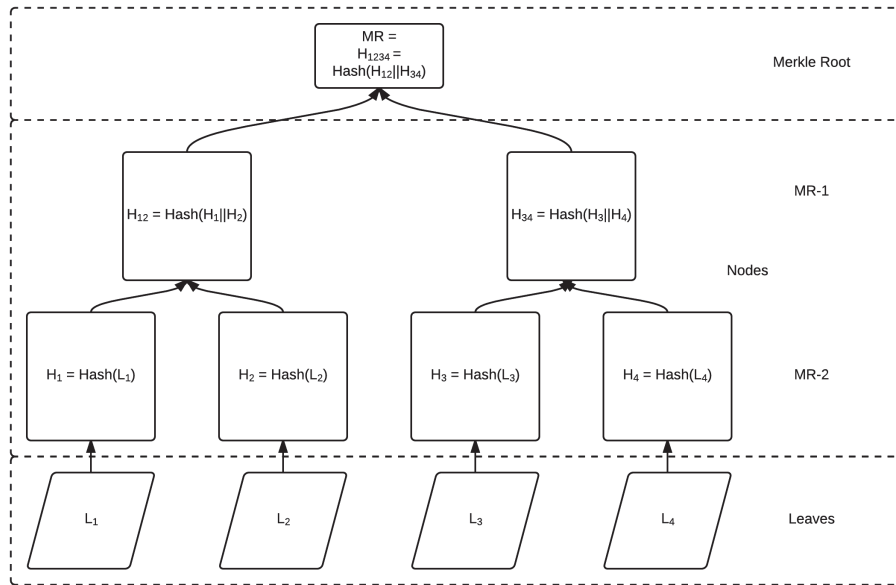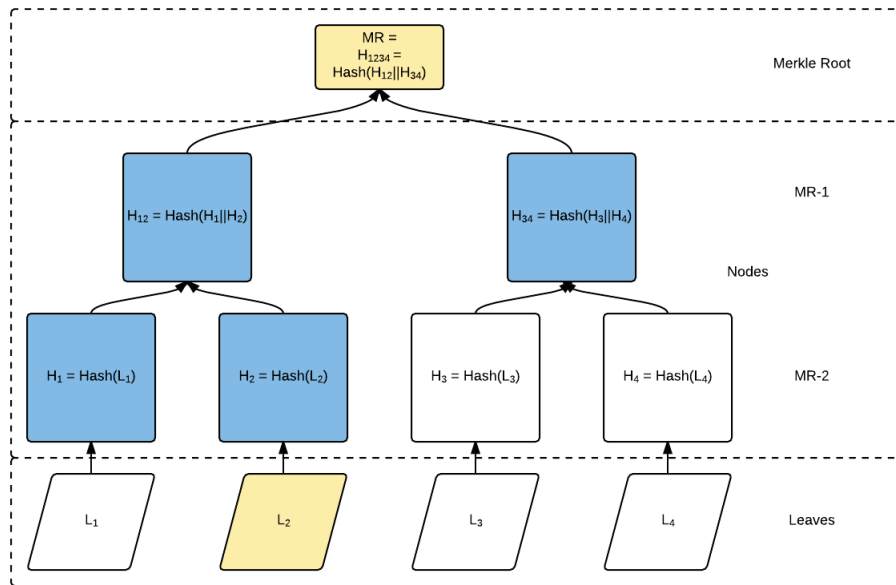
Fig. 3. A symmetric binary Merkle Hash Tree

Fig. 4. A series of relevant OTS commitment operations to verify leaf *L2*

root of that tree is then stored in a block, and returns only the commitments necessary to follow the commitment path up the tree and to the MR.

The OTS timestamp, or proof, is at the core of the OTS protocol. It is the artefact that enables the verification of a given attestation. To understand what a timestamp does, it is necessary to first understand what a timestamp is and what an attestation is. An attestation, in the context of OTS, is a statement that some information - a logical file in the case of the current OTS design - existed in a certain state at a certain point in time. An attestation is, therefore, time-bound and content-specific. An attestation is not a proof in any form

but rather a claim, the authenticity of which is proven by an OTS timestamp.

The timestamp is a series of operations that, when re-played, provides evidence that the attestation is true for a particular source file. The source of truth for OTS is the Bitcoin blockchain, which is demonstrably immutable and chronological, as discussed in section III Since a timestamp is essentially just a collection of commitment operations that are applied to an input message in a specified sequence, replaying those commitment operations in order is all that is necessary to verify the timestamp.

An OTS proof thus allows any person or entity in possession

of the original file or an exact bit-by-bit replica thereof, and the timestamp generated from it, to verify two things without having to trust a third party: that the file existed in a specific time window in the past, and that the file's content remains unmodified from the time the timestamp was created.

When requested to timestamp a file, the OTS client will create a hash of the file and submit it to one or more *calendar servers*. A calendar server adds the file hash as a node in a MHT and provides the MR to the client; it thus *aggregates* hashes into the MHT. A client can also optionally "upgrade" their local proof by requesting the relevant MHT path from the calendar server, thus locally obtaining all the information that is necessary to verify the data; recall that, for a MHT to be verified, the user requires the original data, the MR, and the path through the MHT. After submission the calendar server submits a binary blob representing the MHT to the Bitcoin blockchain; after this point, a client can use the verify operation to verify that the data exists in the blockchain, and obtain the timestamp of that data. The end result is that a large number of hashes can be embedded within the blockchain without incurring a high cost.

A. OpenTimestamps components and trust

To achieve its functional goal, OTS relies on multiple different components, each built on various technologies. OTS was designed to strike a careful balance between ease-of-use and dependencies on systems outside the control of the user.

Due to the nature of OTS and its focus on trust, any system that is not the Bitcoin blockchain or the end-user system introduces a level of uncertainty and potential risk into the OTS timestamp system. Simultaneously, OTS tries to be simple to configure and use to encourage usage; this necessitates that highly technical components can be abstracted and performed on behalf of the user to preserve the user experience. This abstraction leads to the introduction of other systems into the OTS ecosystem. It is, therefore, important that an exploration of these systems is undertaken to understand how they impact the trust placed in an OTS timestamp.

Trust domains – a logical boundary which denotes where a party's control of a particular system begins and ends – are used to better explain OTS components. Recall that OTS attempts to provide easy and trustworthy proofs by eliminating the need for a verifier of a timestamp to trust a third party as trust becomes more fragile as more and more parties are added to the trust chain. It is worth noting then that the failure of any one party will cause the complete trust chain to be broken. This is why OTS attempts to limit the number of systems to trust to the user themselves and the Bitcoin network; essentially, this means dealing with only two trust domains. It is therefore useful to begin by designating three trust domains for explaining various OTS components:

1) SELF: trusted users of OTS and systems in their direct control
2) BTC: the Bitcoin network and blockchain
3) OTHER: neither SELF nor BTC

Ideally, instances where OTHER is trusted need to be avoided where possible. In cases where OTHER cannot be avoided, it is essential to understand how OTHER functions, what protection it provides, and what degree of trust can safely be placed in OTHER without completely compromising the trust of the OTS timestamp.

The **OTS client** is one of the main components in the SELF trust domain, as it is controlled by the user and runs on systems under their control. The libraries and code embedded in the OTS client to interact with the Bitcoin blockchain are therefore also included in SELF trust domain.

The **Bitcoin network** is the only other essential and necessary component of OTS and resides in the BTC trust domain. This domain is considered trustworthy in as far as the Bitcoin network is trusted, underpinned by the resiliency and trust mechanisms which have been discussed previously.

Calendar servers are the only other significant OTS component that potentially fall within the OTHER trust domain. Calendar servers are used to centralise, simplify, and speed up the creation of timestamps at the cost of delegating some trust to the OTHER domain. These are used to provide aggregation services, blockchain interactions services and attestation services for users who choose to, or cannot, run these services locally. Note that the use of calendar servers is not required and that OTS, if configured to do so with the installation of the necessary Bitcoin services, can directly interact with the Bitcoin blockchain to create and verify timestamps.

Calendar servers are not necessarily in the OTHER domain since they can be run privately by the user if they choose to centralise the aggregation and blockchain interaction within the SELF trust domain. One may, for example, think of a company providing OTS calendar servers as part of a private OTS notary service.

The default OTS configuration, as used for illustrative purposes in this work, relies on three public calendar servers:

- https://a.pool.opentimestamps.org Alias: https://alice.btc.calendar.opentimestamps.org
- https://b.pool.opentimestamps.org Alias: https://bob.btc.calendar.opentimestamps.org
- https://a.pool.eternitywall.com Alias: https://finney.calendar.eternitywall.com

These public calendar servers are maintained by the creators of OTS and are used by default in OTS to allow the easy creation of OTS timestamps by foregoing the need for the user to install, configure and maintain a local instance of the necessary Bitcoin software to interact with the blockchain. The installation and maintenance of a full local Bitcoin node can be a daunting task to potential users of OTS, and thus is delegated away from the user and presented as a service in the form of calendar servers. The complexities of configuring, maintaining, and securing a full Bitcoin node is not within the scope of this work.

By using a combination of the defined trust domains and the technology dependencies of OTS to be able to perform timestamps, three distinct configurations (A, B and C) are defined, two of which can be considered fully-trusted (Only SELF and BTC trust domains involved) and the other semi-trusted (SELF, BTS and OTHER trust domains involved). These are illustrated in Table III.

Configuration A, being fully trusted, is depicted in Fig. 5. This configuration requires that user install and run the necessary Bitcoin software on the local environment to enable the OTS client to interact directly with the Bitcoin network.

The configuration depicted in Fig. 5 would require increased effort to configure and run, as all the components would have
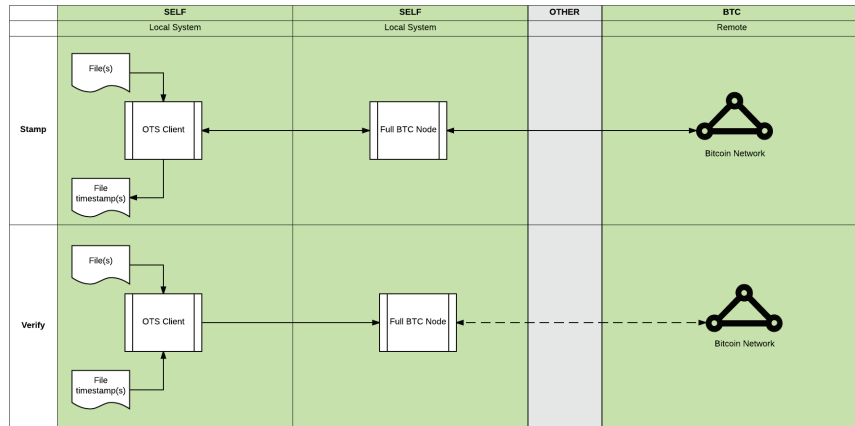
Fig. 5. Major components of OTS in trust domains for Configuration A

TABLE III
RELEVANT TRUST DOMAINS PER CONFIGURATION OPTION

		Trust Domains			
		SELF	*BTC*	*OTHER*	
Config.	A	TRUE	TRUE	FALSE	**Fully-trusted**
	B	TRUE	TRUE	FALSE	
	C	TRUE	TRUE	TRUE	**Semi-trusted**

to be installed by the user. Additionally, this configuration would also carry a cost to the user, since they would be responsible for the transaction fees required to perform the Bitcoin transaction. It is therefore implied that the user would have to have a Bitcoin wallet and a positive Bitcoin balance to successfully interact with the Bitcoin network.

Configuration B, also being fully trusted, is depicted in Fig. 6. This configuration extends the functionality of Configuration A outside the scope of the local system by using a private calendar server. This configuration requires that users install and run a calendar server, as well as install and run the necessary Bitcoin software on the calendar server to enable the OTS client to interact with the Bitcoin network.

By using Configuration B, multiple OTS clients in the SELF trust domain can create and upgrade timestamps without each having to install and run the required Bitcoin services. As with Configuration A, Configuration B would require more effort and skill to configure and maintain while also carrying a cost, in the form of transaction fees, for performing Bitcoin transactions.

Finally, Configuration C, depicted in Fig. 7 is semi-trusted as it includes the OTHER trust domain by making use of public calendar servers. The configuration of C is very similar to B in terms of the required components, the only design change is the fact that the calendar server moves from the SELF to the OTHER trust domain.

By using these public calendar servers, the OTHER trust domain is included in the complete trust chain, and therefore can be considered to be the least trustworthy use case for OTS. It was thought prudent to discuss this configuration, as any other configuration that does not make use of public calendars will be inherently be more trustworthy, and will therefore

only increase the confidence level of the OTS timestamp. Essentially, from a trust and complexity perspective, the worst case scenario for OTS is evaluated. OTS strikes a careful balance between usability and trust, by giving the user the choice of placing their trust only in themselves and the Bitcoin blockchain, or delegating some trust to external OTS systems not controlled by them.

The lifecycle of an OTS timestamp depends heavily on the OTS configuration, since it will determine which systems come into play to create and verify the timestamp. Going forward, the lifecycle of a timestamp is discussed, given OTS is configured as depicted in Fig. 7.

Local dependencies for Configuration C are:

- OTS client: For creating and validating the timestamp and interacting with the public calendar servers.
- Bitcoin node: For verifying the block header in the timestamp.

The above mentioned Bitcoin node can be a pruned node. A pruned node is a node which can function without storing the complete blockchain history with all blocks. A pruned node works by keeping a configurable cache of the latest blocks (specified in MB), thus saving space [22].

Remote dependencies for Configuration C are:

- Public calendar server(s): For timestamp aggregation and interacting with the Bitcoin network.
- Bitcoin network: For storing the data that enables the OTS proof mechanism.

B. OTS functions

A detailed description of the processes and systems involved in each of the core OTS functions will now be presented.

1) Stamping: When stamping a file, the OTS client generates a *SHA256* hash H of the target file. A MHT is then constructed with H to produce a Merkle Root (*MR*). In the case of a single file being timestamped the values of H and MR will be the same, since a MHT with only one value will be the value of the only leaf. If multiple files are timestamped at the same time, the OTS client performs a round of local aggregation by constructing a MHT from the H values of all the files being timestamped to produce a value for MR.
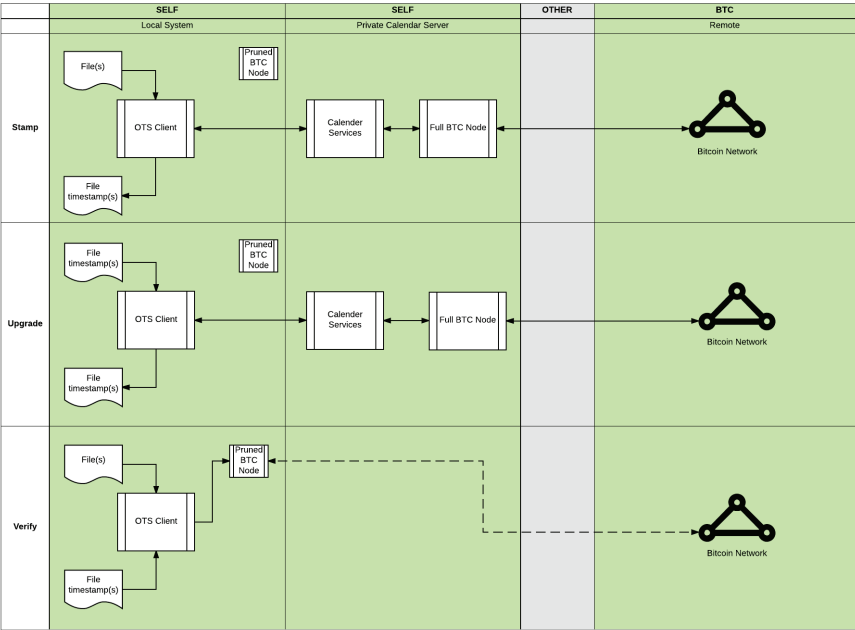
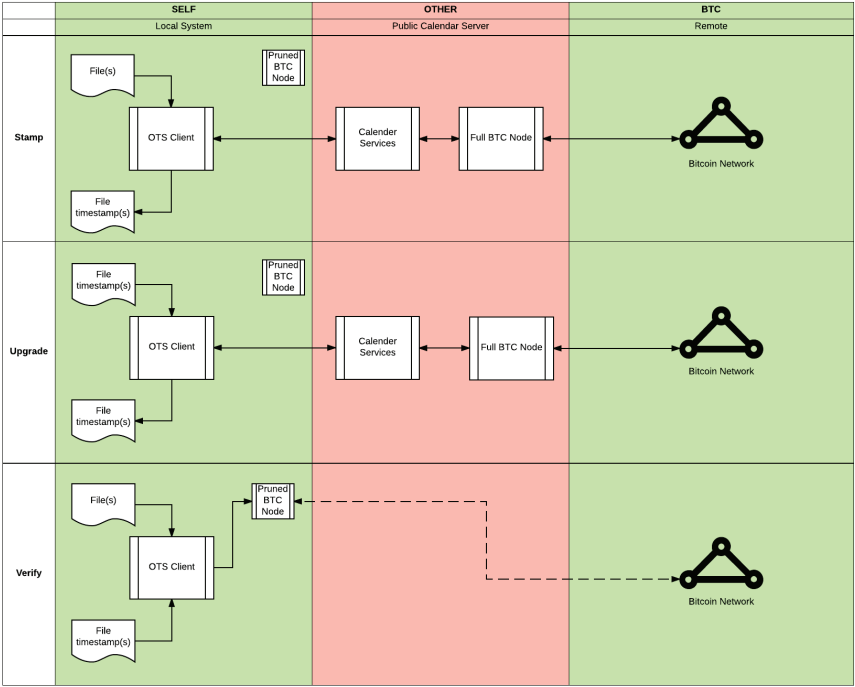Fig. 6. Major components of OTS in trust domains for Configuration B

Fig. 7. Major components of OTS in trust domains for Configuration C

When calculating the MR value, the OTS client appends a random nonce n to the H value of each file. The purpose of this nonce is to preserve privacy, since the MR will be sent to an untrusted public calendar server. The nonce process will be explained in more detail later.

Once the MR value has been derived, an OTS RPC call is made to all the nominated calendar servers supplying the hexadecimal encoded string MR value to the **digest** endpoint. This call is a REST-based web service call over HTTPS and would look similar to the below:

https://[*calender server URL*]/**digest**/[*hex encoded MT value*]

Once the calendar server receives the MR value it performs some validation on the length and structure of the MR value. Upon completion of the validation, the calendar server then performs its own aggregation function by incorporating the MR value into another MHT with all the MR values received from other clients. As mentioned before, this is necessary to make the solution scalable and keep costs low by aggregating many hashes into a single MHT, the MR of which will be embedded into a single Bitcoin transaction as an OP_RETURN opcode.

Depending on the extent of local and remote aggregation, OTS effectively creates nested MHTs where the root of one MHT becomes a leaf in a higher order MHT. This can theoretically be done an infinite number of times to create a single MR from an infinite number of leaves.

Since the calendar server might take some time to aggregate other timestamps and complete the Bitcoin transaction and wait for it to be verified on the blockchain, it cannot synchronously provide the complete proof because the complete timestamp does not yet exist. In lieu of the complete timestamp, the calendar server returns a reduced timestamp which is essentially a commitment that it guarantees it will incorporate the submitted timestamp into a future transaction and return a full timestamp at that point. This is one of the primary examples where trust is placed squarely in the OTHER domain. A malicious calendar server may provide a commitment but discard the timestamp.

It is for this reason that OTS allows the user the ability to submit to multiple calendar servers at the same time while specifying that m of n calendars should return a positive commitment before considering the timestamp submitted. A user also has the ability to provide a whitelist of calendar servers that will be used by the client. If none of those calendars are available, or if the m of n minimum is not met, the timestamp will be considered failed.

Once the incomplete timestamp is received from the calendar server, the OTS client saves the timestamp to the same directory as that of the original file. The returned timestamp will contain the relevant commitment operations and a timestamp identifier for each calendar server that committed to submitting the timestamp.

Once this has been performed the **stamp** process is complete, albeit with a reduced or incomplete timestamp.

2) Info: The simplest of all the OTS functions is the Info function which takes any timestamp as input, parses the commitment operation contained within it and presents them in a legible way to the user. This function is useful if there is a need to see the commitment operation of a particular timestamp or to see if the timestamp is correctly formatted, as any small change in the timestamp will result in a complete parsing failure. The Info function can also be used to determine if a timestamp is complete or if an upgrade request needs to be sent to the calendar server to retrieve the complete timestamp. The Info function operates only locally in the SELF trust domain.

The Info function does not perform any verification of the commitment operations of the timestamp, but only the integrity of the structure of the timestamp.

3) Upgrade: The Upgrade function attempts to upgrade any given incomplete timestamp to a complete timestamp by requesting the complete timestamp from the relevant calendar server(s). A complete timestamp is a timestamp that is locally verifiable without the need to contact a calendar server. Similar to the Stamp function, the Upgrade function needs to interact with a calendar server in the OTHER trust domain, as only the calendar server has the ability to interact with the Bitcoin blockchain. The mechanism for interacting with the calendar server is also very similar, to the digest call, and is performed via an OTS RPC call over HTTPS to a REST endpoint called **timestamp**:

https://[*calender server URL*]/**timestamp**/[*timestamp identifier*]

If the timestamp has been completed by the calendar server, the complete timestamp is returned synchronously to the OTS client as a downloadable binary *.ots* file. Once the OTS client verifies the structure of the timestamp, it proceeds to create a backup of the original incomplete timestamp before appending the .bak extension to it, and merging the complete timestamp into the existing *.ots* file. The OTS client also confirms in the CLI that the timestamp has been upgraded and that it is now a complete timestamp which can be validated locally if a Bitcoin node is present; it then no longer requires interaction with the calendar server.

In the case where an upgrade request is made to a calendar server and the timestamp is not yet complete or was not found on the calendar server, the appropriate message is returned synchronously to the OTS client. Incomplete but found timestamps can again be requested at a later stage by the OTS client.

4) Verify: Verification is the final OTS function, and provides an OTS user the most value by validating the saved timestamp through replaying its commitment operations and verifying the result against the state of the current file. Since it is essential that a very good understanding of how this verification works is obtained, a portion of a manual verification based on the commitment operations contained in the timestamp is conducted.

It is important to note that verification does not necessarily require any interaction with a calendar server if the timestamp has been upgraded. Since verification is such a sensitive and critical operation, OTS was designed in such a way as to ensure it does not require interaction with the OTHER trust domain.

Verification does require that the OTS client be able to query the Bitcoin blockchain for block headers, since the timestamp ultimately points to the block header which contains the transaction which contains the MR derived from the file hash. Verification is performed between the OTS client (SELF) and the Bitcoin blockchain (BTC), by using a locally running Bitcoin node. In the scenario where access to a local Bitcoin

node or one in the SELF domain is not possible, the timestamp can still be verified by contacting the calendar server, however that necessarily weakens the proof as the OTHER domain is involved in attesting to the validity of the timestamp.

There is a significant difference in size and complexity between an incomplete timestamp and a complete timestamp. This difference is a direct result of the Upgrade function, since the entire timestamp and all relevant commitment operations have been retrieved from the calendar server. This would include commitment operations for local aggregation, calendar server aggregation, and the Bitcoin transaction itself.

Note that although there may be multiple distinct commitments, due to the initial timestamp being submitted to multiple calendar servers, the complete timestamp only needs to be retrieved from a single source. Retrieval of the complete timestamps from other sources is pointless since a single valid timestamp is sufficient to perform local verification. Verification results in attestation, which is a statement that confirms that a particular block, with a particular block creation time, does contain the specified timestamp.

Below is a step-by-step walkthrough of exactly how a timestamp is verified, and how it is possible to make an attestation. For the sake of brevity each commitment operation in the complete timestamp will not be manually reproduced. Rather, select examples will illustrate how that can be done. Since the hashes can become very long, the example hexadecimal data below has been rendered using ASCII85 encoding [23] to reduce the amount of space required for display.

1) Compute the SHA256 hash of the data.
   A=JKI*drCOFmd:PVZ:JOPJf¡!.=;BT'e!ˆNaˌXi+

   When run from the command-line, the first step the OTS client performs is to look up the original file based on the timestamp name. If the file is found in the same directory, it performs a SHA256 hash of the file. This hash value serves as the starting point for the timestamp verification and is the first commitment in a series of commitments.

2) Local noncing.
   A=JKI*drCOFmd:PVZ:JOPJf¡!.=;BT'e!ˆNaˌXi+Yt4S:fY@b(7ˆ89&ˌ,ˆ5n

   Due to the privacy concerns of sending the hash of a potentially sensitive file to an untrusted calendar server, the OTS client appends a 128bit random nonce.

3) Re-hashing after noncing.
   ]e#&"K"12@¡))0tc6AkP,ˆC.U+7W%";8uOB('(]L

   The concatenated value is then hashed again. This hides the nonce from being viewed by any other party. The user must retain the nonce value to be able to prove that the calendar server has committed the stated data.

4) Submission and time encoding.
   =jtVL'tc5!jci7HCeH!82D"j"kN92?H=B˙p0g'ai%˙Kbf

   The value is now sent to the calendar server, which prepends the system time on the server. This value is **not** reliable since it is entirely dependent on the system clock of the calendar server and the trustworthiness of the calendar server. Nevertheless, it gives some indication of when a submission may have been made.

5) Method authentication code.
   =jtVL'tc5!jci7HCeH!82D"j"kN92?H=B˙p0g'ai%˙Kbf:t+#&:jIlg

   The server then generates a hashed message authentication code (HMAC) based on the time and a secret key

that the server holds. This HMAC can be used to state authoritatively that a particular calendar server handled the message. The HMAC is appended to the message.

6) Re-hashing and reply.
   ¡]5)'c"KdU.KeiS2)00O+nA#@TD;R3)r$NlY0rT1

   The value is now hashed again, using SHA256, and the entire sequence of operations (including prepended value, appended value, and hashed result) are send back to the client. The client is free to re-calculate the hashed result using the prepended and appended values that are supplied to it. The hashed result is what will be aggregated into the MHT, the Merkle Root of which will be entered into the blockchain.

When submitted as a transaction to the Bitcoin network, one input and two outputs will be seen. The first input and output are used to provide a transaction fee, and thus incentivize miners to include the transaction. The second output stores the MR within its *scriptPubkey* field (see Table II), and precedes it with a Script instruction that makes it provably unspendable under any circumstances. The entire transaction is approximately 150 bytes, which provides a further incentive to miners to include it since it does not take up too much valuable space.

An upgraded timestamp provided by the calendar server will include the submitted hash, all of the necessary hashes that lead up towards the MR, and the transaction hash that uniquely identifies the transaction on the blockchain. It is then possible to look up the transaction on the blockchain independently of the calendar server and verify that the provided operations and hashes do lead up to the blockchain-stored MR. The OTS client does this as part of its verification functionality. It is important to note that the attested time that the OTS client gives is, in fact, derived from the *nTime* field of the transaction's block, and not related to the prepended time used by the calendar server.

By trusting the Bitcoin network with its inherent integrity and immutability, assurance is established that this timestamp cannot be forged and that the contents of the block also cannot be forged or altered. And since the file hash indirectly exists in a confirmed transaction output script in that block, it is known that the file that produced that hash must have existed on or before the stated date.

VI. Testing

Testing of two aspects of OTS was conducted in order to verify its usefulness in a digital forensics context: the timing and accuracy of the timestamping, and the failure rate of OTS. Data on the former was obtained by using the OTS client to stamp, submit, upgrade, and verify timestamps, and noting how long each of these operations took. The failure rate was obtained by intentionally tampering with OTS artefacts to create invalid files and timestamps, and attempting to verify them using OTS.

Data gathering lasted for 34 days, from 5 September 2017 up to and including 8 October 2017. OTS timestamps were created, upgraded, and verified every 10 minutes, resulting in a data set of 4 702 unique files, their timestamps, timestamp results, and operational metadata. These were then analyzed to derive the following values:

- *tStamp*: the time in seconds that it takes to create a timestamp and get a commitment from the calendar server, including local processing time.
- *tUpgrade*: the time in seconds that it takes to upgrade a timestamp to a complete timestamp, including local processing time.
- *tVerify*: the time in seconds that it takes to verify a timestamp, including local processing time.
- *tAccuracy*: the time difference in seconds between the time the timestamp was completed (timeToUpgrade as dipicted in Fig. 9) and the time attestation received by the Bitcoin blockchain as per the OTS verify operation.

Supplementary to the base data set, some metadata about OTS operations was captured by calculating the start and end times of each OTS operation performed by the script. These measurements aimed to accurately measure the execution time of these OTS operations. Initially, this was simply for potential troubleshooting, but it became clear that having a data set of OTS operation times could be valuable and that this data set was also analysed. The times taken to verify operations (*VerifyOperationTime*) and stamp operations (*StampOperationTime*) were recorded.

A final data set was gathered pertaining to the failure rate of OTS. The data set was generated by intentionally tampering with OTS components to induce invalid files and timestamps and reverifying them using OTS. Modification and validation were performed using a Python script, which also recorded the results. The script enumerated all of the previously generated files and timestamps, and alternated between modifying the files, or the associated timestamp, by appending a few fixed bytes. By intentionally breaking the timestamps, or modifying the files in known and consistent way, more insight into potential false positive and false negative results from the OTS Verify function can be gathered.

Using the above-mentioned data sets, more in-depth analysis was performed on each data set to highlight trends, issues and other potentially significant facts.

A. Data analysis

For each of the *tStamp*, *tVerify*, *tUpgrade*, and *tAccuracy* metrics, an average, minimum, maximum, and standard deviation was calculated. These values are listed in Table IV.

All of the measurement values in Table IV are rounded up to two decimal places. These values will henceforth be referred to by concatenating the names of the relevant row and column, e.g. the Average (A) timeToUpgrade (tUpgrade) will be denoted by *A-tUpgrade*.

The time to create a complete timestamp has been visualised in Fig. 9. On the x-axis is the creation time of the timestamp (proofCreatedTime), and the time the proof was created (timeToStamp) is on the y-axis. Additionally, there is a calculated moving average per 144 data points (1 day) to assist in visualising the timestamp completion-time without some of the outlier values. The overall average for timeToUpgrade (*A-tUpgrade*) is 3 563.04 seconds, as can be seen in Table IV.

Similarly, Fig. 11 shows the timestamp accuracy. Timestamp accuracy is defined as the difference in time between the point the timestamp was created (the known time data existed and was committed), and the time verification can attest the data first existed. This is used to measure accuracy as it shows

how precise OTS attestations are for a sample with a known creation data.

A moving average over 144 data points is also calculated and shown in Fig. 11 to account for outliers with the overall average *A-tAccuracy* being 2687.64 seconds. Both of these metrics visualised in Fig. 9 and Fig. 11 are relevant to the responsiveness and performance of OTS within the test environment.

Another aspect of OTS performance is the time it takes to perform individual granular functions. Granular functions refer to the actual time taken to perform a single operation, i.e. Stamp or Verify. Previous measurements were related to the time between multiple operations i.e., Stamp and Verify. The following data set relates to the time it takes to perform individual functions or the time it takes for OTS functions to deliver a requested result.

Granular execution times were recorded for OTS functions:

- *tStampG*: Stamp (All stamp actions including RPC call to remote calendar).
- *tVerifyG*: Verify (All verify actions including RPC call to local Bitcoin node)

Table V shows a summary of a few metrics about execution time of specific OTS operations that were measurable as part of the testing design.

All values in Table V are measurements of seconds taken, rounded up to two decimal places. As with the preceding data sets, the sample covered all of the files created during the test window.

The final piece of analysis was performed sought to evaluate the error rate of the OTS Verify function. The Verify funtion, as stated previously, is the most critical of all OTS operations as it is the mechanism by which OTS delivers the attestation result. It is also likely that Stamp and Upgrade may only be executed once for any particular file, but that Verify might be executed many times throughout the relevant life of the file.

Table VI shows a summary of the results of this analysis phase.

VII. DISCUSSION

1) Observations and interpretation: In this section, the results of the various analyses performed in Section VI-A are discussed and key observations are drawn from the results.

a) Performance and timing: Looking at Table IV, *A-tStamp* execution time is 2.13 seconds on average, with a minimum execution time (*Mi-tStamp*) of 1.18 seconds and a maximum execution time (*Ma-tStamp*) of 4.98 seconds; this is particularly significant and will be elaborated on further in the following section. It should be reiterated at this point that the actions in *tStamp* includes the hashing of the file, noncing, RPC call to a remote calendar server, and the response written to the local disk. Lastly, the standard deviation at (*S-tStamp*) is 0.34 seconds, showing a tight clustering around the mean, and demonstrating consistency under test conditions. These values tend to be low and consistent as the Stamp operation only submits the timestamp to the calendar server and there is no interaction with the Bitcoin blockchain at this point. Higher values for *tStamp* can be expected for larger files, since the SHA256 hash calculation will take longer. The size of files however, will have no noticeable effect of the RPC call

TABLE IV
AVERAGE, MINIMUM, MAXIMUM, AND STANDARD DEVIATION

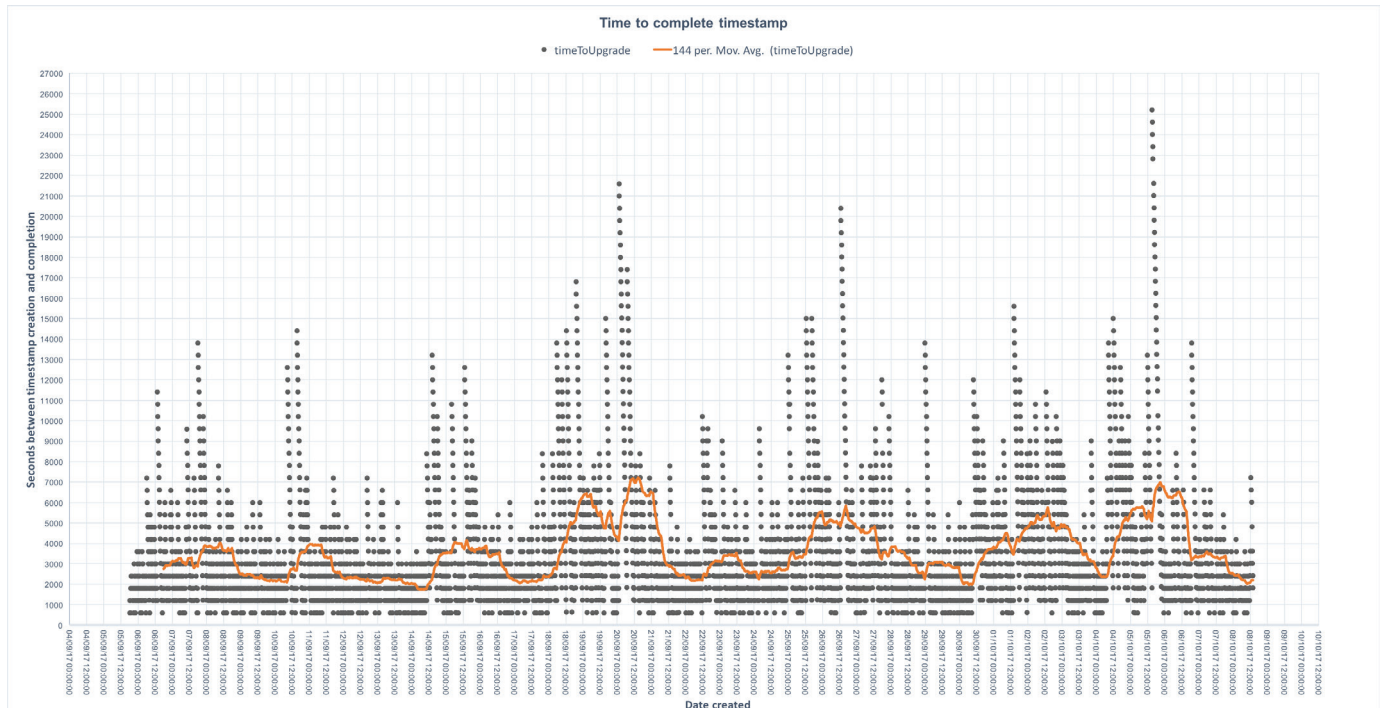|	tStamp	tUpgrade	tVerify	tAccuracy
Average (A)	2.13	3563.04	9.40	2687.64
Minimum (Mi)	1.18	600.92	1.36	21.90
Maximum (Ma)	4.98	25208.67	72.79	24568.47
Standard Deviation (S)	0.34	3105.17	7.66	3074.74

Fig. 9. Time, in seconds, to complete a timestamp relative to the date and time the timestamp was created.
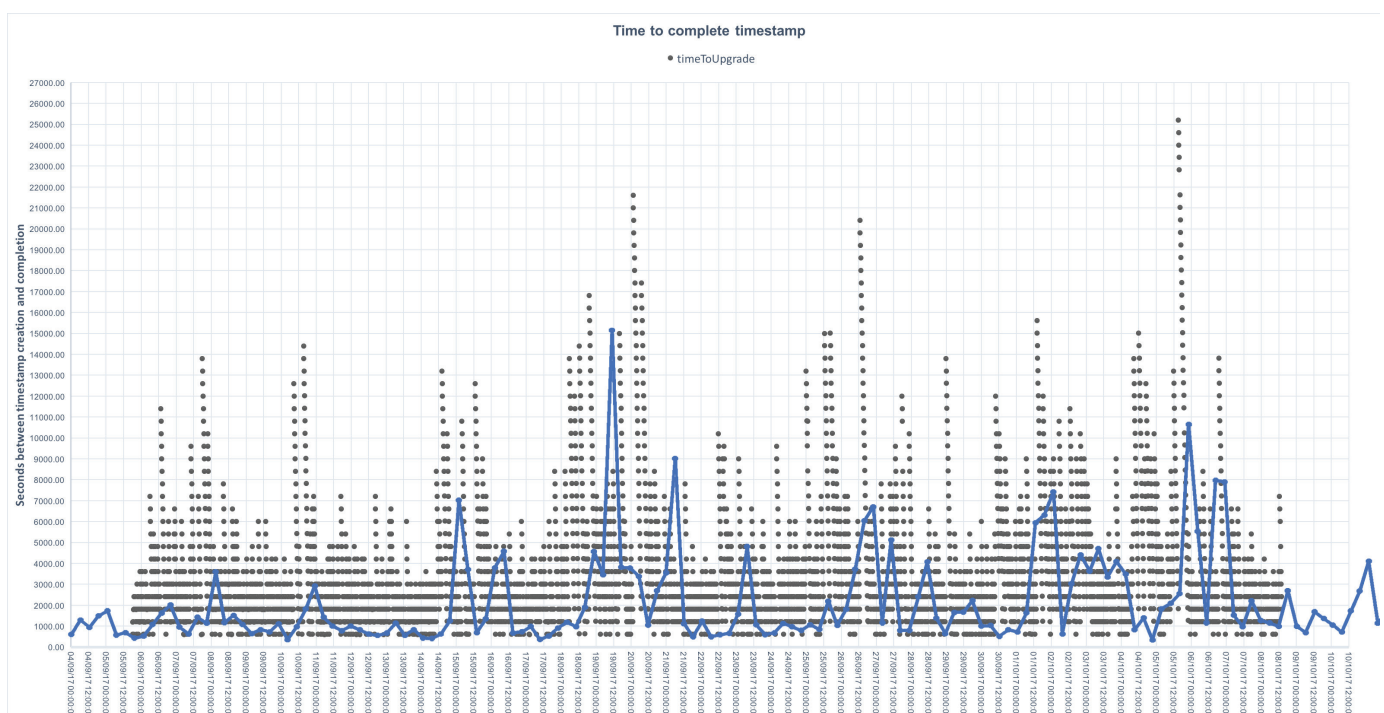

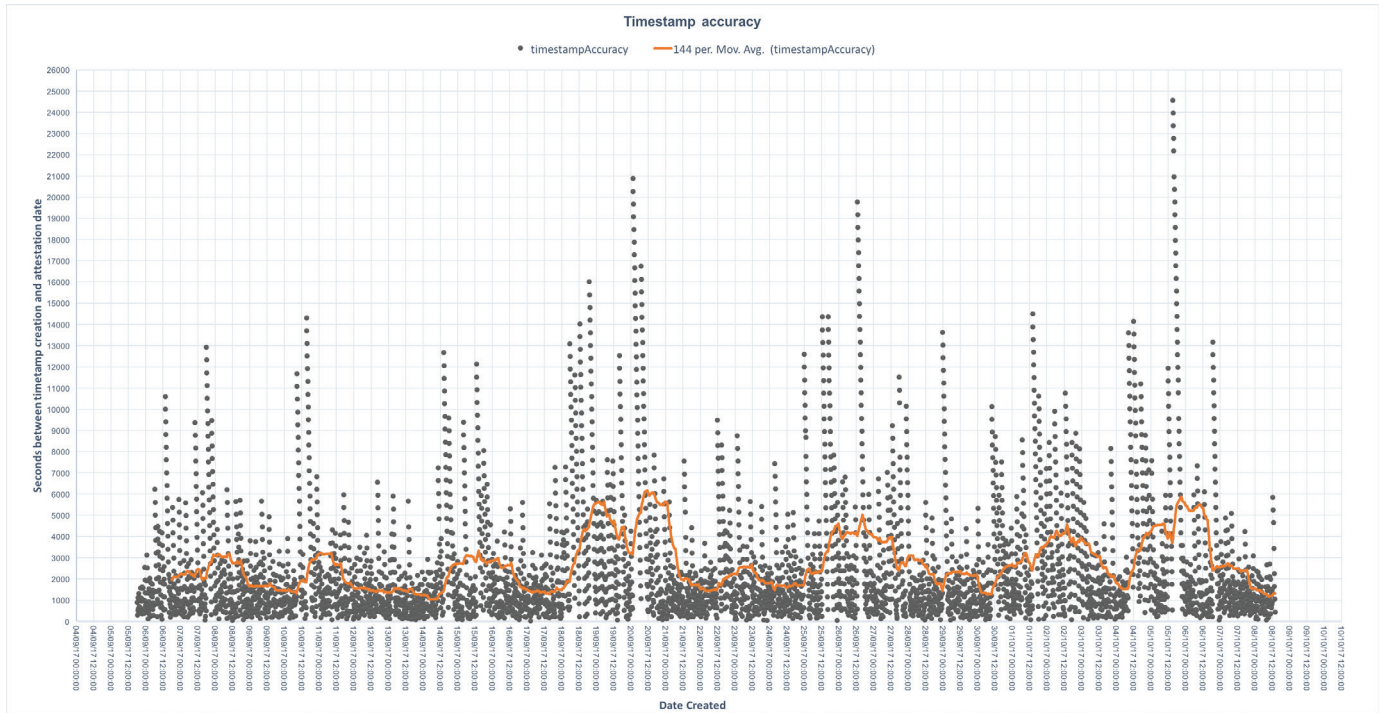Fig. 10. Overlay of average block confirmation time from [24] onto Fig. 9

Fig. 11. Accuracy of a timestamp, in seconds, relative to the date and time the timestamp was created.

TABLE V
GRANULAR OTS FUNCTION EXECUTION TIME

	tStampG	tVerifyG
Average (A)	2.11	0.26
Minimum (Mi)	1.22	0.16
Maximum (Ma)	8.67	13.64
Standard Deviation (S)	0.50	0.23

and response as the data being transferred is a fixed-length SHA256 hash.

With regard to tUpgrade, *A-tUpgrade* is shown to have a much larger value at 3 563.04 seconds; this represents the time between two asynchronous, but related, operations. *Mi-tUpgrade* is measured at 600.92 seconds, representing an instance where the timestamp was upgraded to a complete timestamp in approximately 10 minutes. The actual time to upgrade to a complete timestamp could actually be an even lower value, but due to the test script execution scheduled every ten minutes, the best-case scenario would be 600 seconds plus some processing time for the RPC call. *Ma-tUpgrade* is high at 25 208.67 seconds or approximately 7 hours. High values like these are a side-effect of slower Bitcoin network performance and higher block confirmation times, as shown in Fig. 10, where the six hour moving average block confirmation time for the same period is displayed on top of the data set. With an average block confirmation time of 10 800 seconds, approximately the same time of *Ma-tUpgrade*, it is entirely possible that *Ma-tUpgrade* could have even higher values as it may have been included in a block with a higher than average block confirmation time. *S-tUpgrade*, at 3 105.17 seconds, is close to the value of *A-tUpgrade*, and shows more variation, but none the less, consistent with times of operations between

only the OTS client and public calendar servers.

tVerify, being the time it takes to verify a timestamp inclusive of local processing time, has an average value (*A-tVerify*) of 9.40 seconds; a minimum value (*Mi-tVerify*) of 1.36; and a maximum value (*Ma-tVerify*) of 72.79 seconds. The standard deviation (*S-tVerify*) is 7.66 seconds, showing as with other, primarily local operations, a close clustering around the mean and consistent performance. The slightly higher value of *tVerify* compared to *tStamp*, which is also a local operation, can be attributed to the fact that, during the Verify function, all of the commitment operations in the complete timestamp need to be replayed and the result verified against the local Bitcoin node.

The value of *tAccuracy* is referred to as the timestamp accuracy, and is a measurement of the time difference between the time the initial timestamp commitment was received from a remote calendar server (the date and time the attestation was requested), to the time the Bitcoin blockchain can attest the data existed. The shorter the time span, the more accurate the timestamp attestation can be considered. Accuracy is very closely tied to the block confirmation time, as noted in Section V-A, and thus heavily influenced by the Bitcoin network performance. *A-tAccuracy* is 2687.64 seconds, which is slightly lower than *A-tUpgrade*, as it excluded any lag induced by the script execution timing. *Ma-tAccuracy* is 24 568.47 seconds and *Mi-tAccuracy* is 21.90 seconds, both close to but slightly less than *Ma-tUpgrade* and *Mi-tUpgrade*. *Mi-tAccuracy*, at 21.90 seconds, is an example of where a timestamp was requested from the remote calendar server close to the end of its aggregation cycle and transaction creation in the Bitcoin blockchain. It is also clear that the block confirmation time around the submission of transaction containing the data from *Mi-tAccuracy* would be very short. *Mi-tAccuracy* was committed to the calendar server at 17/09/17 02:20:04, incorporated

TABLE VI
ERROR RATE OF VERIFY FUNCTION

Number of files tested	Pre-modification result		Modify action	Post-modification result		False positive result	False negative result
	True	False		True	False		
2351	2351	0	Modify file	0	2351	0	0
2351	2351	0	Modify timestamp	0	2351	0	0

into the aggregated MHT of which the root was incorporated into a Bitcoin transaction shortly afterward. This transaction was contained in a block which was confirmed by 17/09/17 02:20:26. This very low block confirmation time is supported by the data in Fig. 10, which indicates that the average block confirmation time at its approximate creation time was about 500 seconds. *S-tAccuracy* is close to *A-tAccuracy* which indicates similar clustering around the mean to *S-tUpgrade*.

*tStamp*G and *tVerifyG* are very granular measurements of the OTS functions without local processing time performed by the script like file creation or enumeration. *A-tStampG*, at 2.11, is slightly lower than *A-tUpgrade*, indicating a script processing time overhead of 0.02 seconds. Similarly, *A-tVerifyG* shows very fast execution times of 0.026 seconds as a result of it being a local operation between the OTS client and the local Bitcoin node. *S-tStampG* is very low, at 0.50 seconds, showing very tight clustering around the mean and very consistent performance. *S-tVerifyG* is close to *A-tVerifyG*, which indicates slightly less consistent execution times than that of *A-tStampG*.

In Fig. 11, there is an instance between the dates 05/10/17 00:00:00 and 06/10/17 00:00:00 of an apparent cascading effect from very high y-axis values to lower values, with fixed intervals on the Y-axis. This cascading effect, along with others on Fig. 11, can be explained by slow block confirmation times during those dates. Regardless, the script that created the file and submitted the timestamp executed every 10 minutes irrespective of the Bitcoin network performance. This means that as the calendar server aggregates timestamp and waits for a block to be confirmed, multiple timestamps could have been submitted to it. Since the calendar server timing is subject to the block confirmation and the testing script is not, there is a backlog of timestamps being created on the calendar server when block confirmation is delayed. When the block confirmation finally occurs, all of these backlogged timestamps, created over a long time span, are included in the next block. Since this block now contains timestamps created over a matter of hours, ten minutes apart, but has a single confirmation time, the time difference between submission and attestation of the first timestamp submitted is very high and gradually gets smaller for each subsequent timestamp. Looking at Fig. 11, it can be seen that the y-axis values cascade down at intervals of approximately 600 seconds (10 minutes) is indicative of the testing script executing and submitting a new timestamp every 10 minutes. This explanation is further supported by Fig. 10, which clearly indicates higher average block confirmation times around these instances of cascading values.

b) Errors: A very important aspect of this research relates to identifying error rates of OTS as a protocol and implementation. Without known error rates, it is difficult to gauge the level of confidence one can have in OTS. As such, errors rates will be discussed from two perspectives:

- Verification errors: Errors in the verification of valid OTS timestamps.
- Creation errors: Errors in creating a complete OTS timestamp.

Verification errors are the most serious of potential OTS errors, as they indicate that the verification operation does not return a truthful or accurate result. Since the use case for OTS is to reliably get accurate attestations as to the integrity and timestamp of a particular file, any error in this process should be considered serious. Errors in the verification process undermine the fundamental purpose of OTS, and high error rates would indicate that OTS cannot be trusted.

OTS verification was tested extensively over the entire test sample. Exactly half of the test sample (2 351), previously validated, was modified by appending a few fixed bytes to the original file. The other half were modified by appending the same fixed bytes to the timestamp associated to the file. In this way, the error rates for invalid files or invalid timestamps, both of which should result in an outright failure to verify, were tested.

Errors during this testing could fall into one of two categories: false negative or false positive. False negative results are results where a valid combination of file and timestamp resulted in a failure to verify. False negative results would indicate that either the file integrity was compromised, or the timestamp changed when it was not the case. False positive results are results where an invalid combination of file and timestamp (either modified) resulted in a positive verification result, indicating that the file integrity is sound or that the timestamp is valid when it is, in fact, not the case. A false negative result would result in OTS users trusting files and attestations that are not correct.

As can be seen in column 5 in Table VI, there were no instances of false positive OTS verifications for the tested sample. Similarly, in column 6 it can be seen that there were also no false negative OTS verification results in the tested sample. With zero errors in the tested sample, it is clear that the robust verification mechanism and the fragility of the timestamp structure combine to create a very reliable system with a near 0%, or insignificant, failure rate.

Creation errors are the second type of error, and relate to any error in creating a complete OTS timestamp for a file. This type of error is less serious than a failure to verify, as it
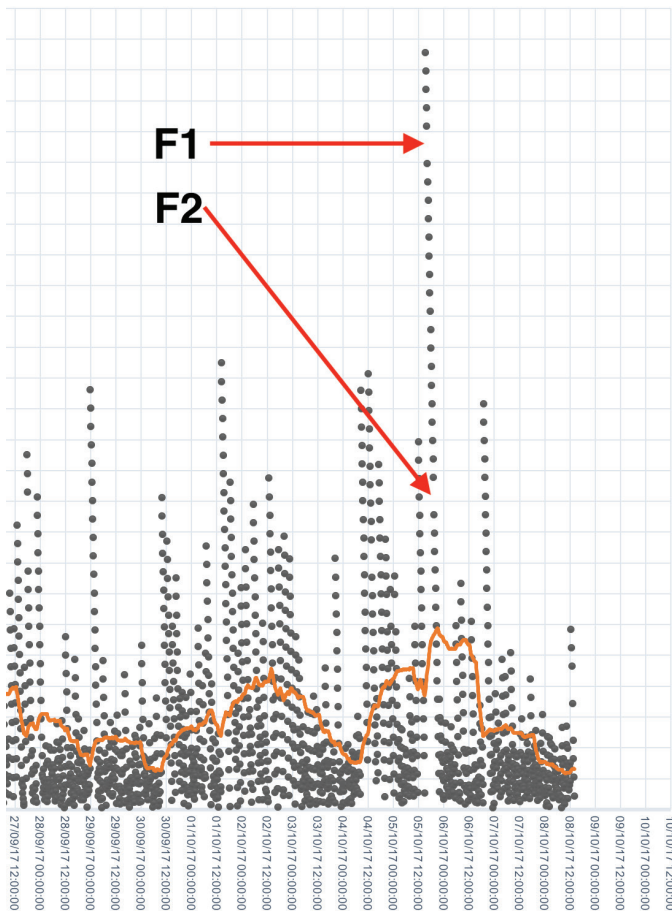
Fig. 12. Missing timestamps results due to a failure to create timestamp.

does not make any claim toward the integrity of the file. The risk with a failure to create a file is that some critical data or evidentiary item may not receive a complete timestamp and could therefore not be validated at some future point in time when its integrity is questioned. During the testing process there was no specific test for a failure to create a timestamp, but as data was being analysed and correlated along with metadata from the script logs, it became apparent that a small number of timestamps failed to complete.

The log entries were created when the Upgrade function was attempted, but failed for a particular file. They indicated that the Upgrade function could not find the timestamp file for the file in question. There were 26 instances of such errors in the tested sample. Manual verification confirmed that all 26 files did not have any timestamp file and there was no indication that a timestamp file existed at any point in time, despite the script log confirming a timestamp was created. There were also no errors in the logs indicating that the OTS Stamp function failed. The only common factor between all 26 instances of these failures, were that the stamp creation time logged in the script execution was higher than the maximum stamp time *Ma-tStamp* allows for valid timestamp. These logs did not reveal a root cause for this failure, but by a process of elimination it was determined that the failure either occurred in the RPC call to the calendar server, or locally when the returned timestamp commitment was saved to disk. Unfortunately, there were no logs to indicate which of the two

processes were malfunctioning.

Instances of such failures to create a timestamp can be seen in Fig. 12, where there are clear gaps between two data points.

Being unable to isolate the root cause to an OTS- or operating system-specific failure, it is necessary assume the worst-case scenario from the perspective of OTS, which is that the calendar server did not return a valid timestamp commitment for the 26 stamp requests. Even so, the number of failures as a proportion of the overall data set is extremely small at 0.553%. Even though the timestamp failed to create when requested, this does not prevent the request from simply being resubmitted if the failure is detected. It is, therefore, suggested that all timestamp creation be immediately validated when OTS is used in potentially significant use cases, such as a digital forensic investigation relating to legal proceedings.

VIII. CONCLUSION

This paper has described the field of digital forensics, its requirements, blockchain technology, the interaction between these, and independently analysed promising open source software in this niche. The results have been impressive, and certainly merit a great deal of discussion in the field of digital forensics. Blockchain technology, coupled with the design of OTS, has the potential to take one more human factor out of the digital forensics equation and increase trust in digital evidence as a whole. One caveat, however, is that timestamp accuracy is not to be taken for granted. The blockchain may therefore not be appropriate for digital evidence that must be independently timestamped with great accuracy.

REFERENCES

[1] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem," *ACM Trans. Program. Lang. Syst.*, vol. 4, no. 3, pp. 382–401, Jul. 1982. [Online]. Available: http://doi.acm.org/10.1145/357172.357176

[2] S. Nakamoto, "Bitcoin: A Peer-to-Peer Electronic Cash System," 2008. [Online]. Available: https://bitcoin.org/bitcoin.pdf

[3] Ethereum Foundation, "Ethereum," 2016. [Online]. Available: https://www.ethereum.org/

[4] Zerocoin Electric Coin Company, "About Us," 2016. [Online]. Available: https://z.cash/about.html

[5] B. Nelson, A. Phillips, and C. Steuart, *Guide to Computer Forensics and Investigations*, 5th ed. Delmar Learning, 2015.

[6] A. Valjarevic and H. S. Venter, "Implementation guidelines for a harmonised digital forensic investigation readiness process model," *2013 Information Security for South Africa - Proceedings of the ISSA 2013 Conference*, pp. 1–9, aug 2013.

[7] J. Dykstra and A. T. Sherman, "Acquiring forensic evidence from infrastructure-as-a-service cloud computing: Exploring and evaluating tools, trust, and techniques," *Digital Investigation*, vol. 9, pp. 90–98, 2012.

[8] C. Wilson, "Digital Evidence Discrepancies: Casey Anthony Trial," 2011. [Online]. Available: http://www.digital-detective.net/digital-evidence-discrepancies-casey-anthony-trial/

[9] J. H. Witte, "The Blockchain: A Gentle Introduction," pp. 1–5, 2016.

[10] B. Preneel, "Cryptographic hash functions," *European Transactions on Telecommunications*, vol. 5, no. 4, pp. 431–448, 1994.

[11] Y. M. Motara, "Preimages for SHA-1," Ph.D. dissertation, Rhodes University, 2017.

[12] G. C. Kessler, "Anti-Forensics and the Digital Investigator," *Proceedings of the 2014 47th Hawaii International Conference on System Sciences*, pp. 1–7, 2006.

[13] P. Rogaway, *Nonce-Based Symmetric Encryption*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 348–358.

[14] B. Schneier, *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. New York, NY, USA: John Wiley & Sons, Inc., 1993.

[15] A. Back, "Hashcash - A Denial of Service Counter-Measure," 2002. [Online]. Available: http://www.hashcash.org/papers/hashcash.pdf

[16] K. Okupski, "(ab) using bitcoin for an anti-censorship tool," Ph.D. dissertation, Eindhoven University of Technology, 2015.

[17] B. Gipp, N. Meuschke, and A. Gernandt, "Decentralized Trusted Times-tamping using the Crypto Currency Bitcoin," *iConference 2015*, pp. 1–6, 2015.

[18] V. Wayne, S. Wilkinson, and J. Bukowski, "Chainpoint: A scalable protocol for recording data in the blockchain and generating blockchain receipts," 2016. [Online]. Available: https://tierion.com/chainpoint

[19] M. Araoz and E. Ordano, "Proof of Existence," 2013. [Online]. Available: http://proofofexistence.com/

[20] P. Todd, "OpenTimestamps: Scalable, Trustless, Distributed Timestamping with Bitcoin," 2016. [Online]. Available: https://petertodd.org/2016/opentimestamps-announcement

[21] R. C. Merkle, "Protocols for Public Key Cryptography," *Synopsis on Security and Privacy*, pp. 122–134, 1980.

[22] Bitcoin Foundation, "Wallet: Pruning," 2016. [Online]. Available: https://github.com/bitcoin/bitcoin/blob/v0.12.0/doc/release-notes.md#wallet-pruning

[23] Adobe Systems Incorporated, *PostScript Language Reference Manual*, 2nd ed. Addison-Wesley Publishing Company, 1990.

[24] Blockchain Luxembourg S.A, "Average block confirmation time," 2017. [Online]. Available: https://blockchain.info/charts/avg-confirmation-time?timespan=60days&showDataPoints=true

William Thomas Weilbach Thomas, who started his career as a software developer, quickly gravitated to information security after being exposed to application security. After completing his postgraduate research on the topic of digital forensics, he made information security his primary professional focus as an information security specialist at a major South African bank. Thomas then completed his MSc in Computer Science at Rhodes University where he continued his work on digital forensics and graduated with distinction in 2018, having been awarded the MWR prize for the Best Information Security Student. Presently Thomas continues his passion for information security, but returned his focus to Application Security by enabling the same at a major sovereign wealth fund.

Yusuf Moosa Motara Yusuf Motara is a Senior Lecturer in Computer Science at Rhodes University. His interests are functional programming, information security, software development, and computer science education. He is presently working on the modeling of functional programs. Dr Motara lives in Makhanda with his wife, children, and a great deal of contentment.