Practical Application of Distributed Ledger Technology in Support of Digital Evidence Integrity Verification Processes

Submitted in partial fulfilment
of the requirements of the degree of

Master of Science

of Rhodes University

William Thomas Weilbach

Grahamstown, South Africa
December 11, 2017

Abstract

After its birth in cryptocurrencies, distributed ledger (blockchain) technology rapidly grew in popularity in other technology domains. Alternative applications of this technology range from digitizing the bank guarantees process for commercial property leases (Anz and IBM, 2017) to tracking the provenance of high-value physical goods (Everledger Ltd., 2017). As a whole, distributed ledger technology has acted as a catalyst to the rise of many innovative alternative solutions to existing problems, mostly associated with trust and integrity.

In this research, a niche application of this technology is proposed for use in digital forensics by providing a mechanism for the transparent and irrefutable verification of digital evidence, ensuring its integrity as established blockchains serve as an ideal mechanism to store and validate arbitrary data against.

Evaluation and identification of candidate technologies in this domain is based on a set of requirements derived from previous work in this field (Weilbach, 2014). OpenTimestamps (Todd, 2016b) is chosen as the foundation of further work for its robust architecture, transparent nature and multi-platform support. A robust evaluation and discussion of OpenTimestamps is performed to reinforce why it can be trusted as an implementation and protocol.

An implementation of OpenTimestamps is designed for the popular open source forensic tool, Autopsy, and an Autopsy module is subsequently developed and released to the public. OpenTimestamps is tested at scale and found to have insignificant error rates for the verification of timestamps. Through practical implementation and extensive testing, it is shown that OpenTimestamps has the potential to significantly advance the practice of digital evidence integrity verification. A conclusion is reached by discussing some of the limitations of OpenTimestamps in terms of accuracy and error rates. It is shown that although OpenTimestamps has very specific timing claims in the attestation, with a near zero error rate, the actual attestation is truly accurate to within a day. This is followed by proposing potential avenues for future work.

Acknowledgements

To my wife, Liezel, thank you for your loving assistance, understanding and relentless support. Thank you for the motivation you provided when it was most necessary. I could not have completed this without you.

To my mother and father, thank you for showing me the value of hard work and for instilling in me an insatiable thirst for more knowledge.

To my sister and brother, thank you for setting a great example, for providing the motivation to always strive to do better and to never stop exploring or asking questions.

To Jock, the trailblazer, thank you for everything you have done for me in the past few years both professionally and personally.

To my supervisor, Yusuf, thank you for all the guidance you provided throughout this process.

To the open source software community, thank you for enabling all of this and so much more.

Finally, to all my friends and family, thank you for understanding and tolerating far too many absences. I'm back now.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

In the face of an impending financial crisis, an anonymous researcher, going by the pseudonym Satoshi Nakamoto, proposed a mathematical solution to the problem of decentralised and distributed trust. This solution, in the form of blockchain technology, was presented by Nakamoto (2008) in a paper titled: "Bitcoin: A Peer-to-Peer Electronic Cash System". Blockchain technology, a form of applied cryptography at the core of Bitcoin, has emerged as a significant and potentially revolutionary technology inspiring a new class of solutions to problems all but forgotten. This proposition by Nakamoto would go on to drastically change how society thinks about money and the financial system supporting its circulation. The core of Nakamoto's idea was adopted and expanded upon with enthusiasm in various other domains and today is more generically known as blockchain.

The potential applications of blockchain technology are vast and continue to diversify every day with the emergence of smart contract platforms such as Ethereum (Ethereum Foundation, 2016), payment solutions such as Ripple[1] and digital currencies such as Zcash (Zerocoin Electric Coin Company, 2016). However, despite its wide adoption, blockchain technology remains relatively unexplored even though the technology demonstrates versatility in areas that extend beyond payments and currency. It achieves this by solving a few

[1] `https://ripple.com/`

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.

It is for this reason that there is an endeavour to innovatively apply blockchain technology to solve issues of trust and integrity in the realm of digital forensics and augment the often-challenged field of digital forensics. By providing a mechanism, process and toolset to formalise the validation of the integrity of digital evidence, a mutual confidence can foster between digital forensics and legal processes.

By leveraging the properties of blockchain technology, a process was researched and then implemented to create an irrefutable, immutable and inherently verifiable proof of the existence of digital evidence. The proposed process is accompanied by a standardised process for performing proof creation and validation in a popular digital forensics software suite.

1.2 Research objectives

The goal of this research is to transparently and conveniently integrate distributed ledger timestamping technology into a digital forensic workflow. To achieve this, the following high level actions were performed:

1. Review of candidate technologies
2. Proof-of-Concept (PoC) implementation of the chosen technologies
3. Measurement and testing of the chosen technology and implementation

Given the pace at which blockchain technologies are being innovated, there is a need to research and clarify how these diverse technologies can be used in the solution postulated. Building such a solution on an ephemeral base technology would mitigate any long-term benefits it could potentially contribute. As an understanding of relevant technologies and their respective benefits and limitations is essential to the ongoing success of this work, a thorough compare and contrast exercise will be conducted to accurately inform the success of the above mentioned goals.

Candidate technologies, starting with previous work by Weilbach (2014) and further informed by newer work like Crespo and García (2017) and Opentimestamps.org (2017),

will be utilised to elicit and develop of a set of requirements and possible limitations. Based on these requirements and limitations produced through the technical review, a chosen candidate technology for interacting with the Bitcoin blockchain will be further researched and evaluated.

This will be followed by a documented PoC implementation of the proposed system, which will be discussed to facilitate a better understanding of its purpose; this PoC will then also serve to inform ongoing work in the field. Focus will be placed on enhanced modularity of the solution as its adoption and subsequent success would be heavily dependent on easy integration into existing digital forensics tools and processes. The implementation of the PoC will serve as a reference implementation to encourage further development of such technologies and their use in the digital forensics domain.

Finally, the reference implementation and the underlying candidate technology will be subjected to rigorous testing to measure its effectiveness and accuracy in aiding digital evidence integrity validation.

1.3 Scope and limits

This work touches on a range of technologies and algorithms associated with applied cryptography and, thus, relies on many cryptographic primitives. The performance and efficiency of these has been tested and evaluated academically (Preneel, 1994) and through repeated real-world application (Käsper, 2012). It is unnecessary to revalidate these well-established results.

The cryptocurrency ecosystem, especially Bitcoin, happens to be in a state of significant change as a result of the ongoing Bitcoin scaling debate (Harper, 2017) at the time of writing. Many proposals are being made that may potentially change the size and structure of Bitcoin blocks going forward. Although a personal opinion on the best solution for the Bitcoin ecosystem is held by the author, that opinion and other potential changes will not be discussed in this research.

The scope of this work will therefore explicitly exclude:

- The efficiency and performance of cryptographic hash algorithms such as *sha256*.
- Effects of changing block size and structure as a result of Bitcoin scaling proposals.

1.4 Document structure

What follows is an outline of the paper structure covering the main themes:

In Chapter 2, a thorough literature study is performed to discuss and analyse the two concepts at the core of this research effort: digital forensics and blockchain technology. As part of this literature study, the basic principles of both concepts are covered to ensure that the reader has the necessary background knowledge. In terms of digital forensics, this paper explores the current tool development practices and challenges faced, specifically relating to evidence integrity and the practices of verifying evidence integrity. In the case of blockchain technology, a discussion on current notable implementations, the limitations of the technology and the future of the technology is presented. Once digital forensics and blockchain technology have been individually covered, a section on past and current research efforts that have linked the two concepts is presented followed by an evaluation of these works for relevance to the stated research goals.

As a result of the evolving, complex nature of the blockchain ecosystem, a detailed compare and contrast exercise between the most prevalent implementations of blockchain timestamping technologies is performed to determine which will be best suited to address the goals of this research. The review will focus on identifying beneficial aspects as well as limiting factors of all current implementations. This section concludes with an informed decision on which system to use as the basis for further development of the PoC implementation.

The design section in Chapter 3 consolidates the knowledge gathered from the literature review to clarify the goals of the research and to propose a research design in support of those goals. The design includes a detailed discussion of the candidate technologies and outlines how these will be integrated in support of the research goals. A final research design, based on a defined research question, includes a review of the timestamping protocol, the chosen method of implementation of the protocol, and a series of tests to measure aspects of the design limitations and benefits of the underlying technologies.

Chapter 4 chronicles the implementation process and presents the design artefacts originating from it. As part of this section, the progress of the implementation in relation to use cases derived from the requirements is shown. Additionally, the choice of implementation language is also discussed in this section, along with challenges relating to the implementation itself inform future development efforts in this field.

Subsequent to the development of the PoC, in Chapter 5, a series of tests to measure the effectiveness and accuracy of the implementation and protocol are performed. These results are measured against set criteria, such as usability, efficiency, speed and interoperability, to develop an educated opinion about the success of the implementation; the analysis performed will be both qualitative and quantitative in nature. Finally, an evaluation of the challenges and limitations of the implementation, informed by results of the various tests, is performed.

In Chapter 6 an evaluation of the overall success of the research effort relative to its stated goals is performed, and the contribution of the work to the field of digital forensics is discussed along with how the use of blockchain technology is essential to the effort. Finally, insights made into the problem domain and possible future work in this problem domain are presented.

Chapter 2

Literature study

Compared to the maturity of research concerning evidence handling and preservation in traditional forensics, the body of knowledge pertaining to digital forensics is in its infancy. In order to contribute to this new, growing body of essential knowledge, a thorough literature review must be conducted to identify what is known and what is unknown. The ultimate goal of improving the admissibility of evidence verified through distributed ledger technologies can only be achieved if there is sufficient understanding of the current research in all relevant domains.

The most relevant domains are digital forensics and distributed ledger te chnologies, the details of which, are essential to understanding the problem and possible solutions to the problem.

The literature review starts with a section on the current state and challenges of digital forensics. In the second section the current state of blockchain technology is reviewed with an emphasis on understanding the emergence of various practical applications of this versatile technology. The final section discusses existing literature aligned with these two previous domains. The literature review is concluded with a section discussing the gaps in the current literature and how these gaps can be addressed.

2.1 Digital forensics

Digital forensics is a science as young as modern personal computing and developed organically from a growing need investigate computer related crime, brought on by the

emergence of cyber crime as networked computing and connectivity became more and more popular in the 1990s (Berghel, 2003). At its advent, it had very limited application and was not widely practiced; this changed at around the turn of the 21st Century with the widespread adoption of networked computing. For the first time ever, it was common practice to share and distribute large volumes of information from person to person, crossing geographical boundaries, using networked computers. The popularisation of networked computing outside of academic circles meant that families, individuals or companies were creating, storing and sharing information of infinite complexity and variation, and each byte of information could potentially become part of an investigation, be it criminal or exploratory. The increased adoption of networked computing also led to a notable increase in digital forensics research, followed by current and sustained growth.

The nature of digital forensics necessitates constant growth and adaptation to accurately deal with the constantly evolving subject matter and operational technology. To date, digital forensics, as a practice and a science, is still playing catch up with the rapid evolution of technology in computing.

Digital forensics, also referred to as computer forensics, deals with the acquisition, storage, investigation and reporting of digital evidence in such a way as to ensure utmost admissibility of the evidence by providing verifiable assurances of its integrity. Tobergte and Curtis (2013) formally define computer forensics as: "...the discipline that combines elements of law and computer science to collect and analyze data from computer systems, networks, wireless communications, and storage devices in a way that is admissible as evidence in a court of law".

2.1.1 Purpose

The digital forensic process, as adapted from Valjarevic and Venter (2013), can, at a high level, be described by three basic practices:

1. Acquisition
2. Analysis
3. Presentation

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 as noted by Dykstra and Sherman (2012). 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, and as Wilson (2011) notes, any issue that adversely affects the admissibility of digital evidence can cast doubt on entire investigations.

The analysis phase can be subdivided into activities such as identification, collection and transportation. With digital evidence, as with physical evidence, the collection and transportation activities pose the greatest threat to the chain of custody, and to the overall integrity of the evidence. Particularly relevant with regard to digital evidence, though, is the inherent need for the evidence to be moved or replicated from its (potentially volatile) source to another system.

Dykstra and Sherman (2012) noted that completeness and accuracy are the two critical measurable attributes of the acquisition phase; they continued by then explaining the complex hierarchy of trust at play during a typical acquisition phase. They noted that trust is required from the network level up to the operating system and application to ensure evidence is free from accidental or intentional tampering. Many tools, techniques and even frameworks have been developed solely for this purpose during the acquisition phase, of which one, aimed at Infrastructure-as-a-Service (IaaS), is discussed in Dykstra and Sherman (2013).

A common, and sometimes mandated practice during the acquisition phase is the act of hashing evidence (Dykstra and Sherman, 2012). A cryptographic hash, also referred to as a digest, is a unique, fixed-length value, generated from any evidentiary artefact of variable length, 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 (Witte, 2016). 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 (Preneel, 1994). 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.

A common use case for the hashing of evidence during initial acquisition would be when a practitioner receives a hard drive disk (HDD) containing potential evidence. They would generate a hash of the contents of the disk, duplicate the disk and then verify the integrity of the copy by comparing the hash of the copy with the hash value of the original. If the two hash values match, verifiable proof exists that the content of the two HDDs are exactly the same. In the above scenario, the responsibility for producing, comparing and verifying the integrity of information is the sole responsibility of the investigator.

The practice of hashing to verify the integrity of evidence is commonplace and may be performed in any scenario where the integrity of the evidence might be questioned. It can be performed on a single artefact, multiple artefacts or fragments of multiple artefacts; the choice is up the investigator. However, the benefit of producing hashes for as much of the digital evidence as is practical is that it would enable the investigator to verify the integrity of the evidence on a very granular level. Hash values and their use will be discussed in much more detail in the coming sections of this work.

As much as the acquisition phase is critical to ensure the successful start of an investigation, the analysis phase is critical to developing a clear picture of events, affirmed with evidentiary artefacts and contextual links. 'Analysis' is a broad term that encapsulates an increasingly large number of specialised practices including, but not limited to, data retrieval, log correlation and exploration.

Kessler (2006) notes that analysis of digital evidence is resource-intensive and usually requires a significant amount of human intelligence over an extended period of time. Garfinkel (2010) explains that the burden of analysis is exacerbated by the fact that, in recent years, an investigation could require analysis of multiple devices as opposed to a single device, as was the norm in years gone by.

The concept of integrity and the chain of custody is as relevant in this phase as in the acquisition phase as there is interaction with the evidence. In an ideal scenario, analysis would not be performed on the original artefacts but rather on a validated copy thereof; I.E., where an investigator receives non-volatile evidence, such as a hard disk, under controlled circumstances and as part of a defined process. In a non-ideal scenario, there would be some level of interaction with the original evidentiary artefact; I.E., where interaction with volatile evidence like memory, is required in the field or in an uncontrolled circumstance. It is during these non-ideal types of interaction that there exists the greatest chance of intentionally or accidentally modifying the evidence in question. Any such modification to the evidence that cannot be explained or reversed, can fundamentally jeopardise the investigation as the integrity of the evidence is immediately questioned.

At a high level, 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 and facts concluded from the previous phases. Depending on the nature of the investigation, the presentation phase could also include a list of necessary actions to remedy an incident or mitigate a vulnerability.

Valjarevic and Venter (2011) noted that during presentation, the following artefacts can

be expected:

- A time-line of events significant to the investigation,
- An explanation of users involved in the incident and their respective relations,
- A time-line of all recorded actions and how these actions relate to users, and
- All irregularities noted during the incident and the investigation.

As can be seen from Kessler (2006), the presentation phase of an investigation can be, and most likely will be, subjected to intense scrutiny regarding the integrity of the processes and evidence. This is especially if the investigation form part of criminal or legal proceedings. 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.

2.1.2 Tool development

As this research includes the development of a toolset that can be used by forensic investigators to timestamp and verify arbitrary evidence signatures against an immutable source, it would be prudent to evaluate the history of tool development in this field.

The development of tools in the digital forensic space is certainly not uncommon. As mentioned before, whole frameworks, protocols and tool suites have been developed to streamline the modern digital forensic process. An example of one such effort is seen in Valjarevic and Venter (2013), with the *Harmonised Digital Forensic Investigation Readiness Process Model*. Examples of popular tools include EnCase, SIFT and Volatility among others (InfoSec Institute, 2017). These tools vary widely in functionality and application, with some targeting very specific problems and others serving as high-level frameworks for performing and managing digital investigations and the associated data. These tools also vary between Free/Open Source Software (FOSS) or Commercial Off-The-Shelf (COTS) solutions.

Digital Forensics, as with almost all computer science disciplines, has had, and continues to have, a strong reliance on FOSS solutions as FOSS continues to support a vast section of the information systems advanced users and the public rely on every day. It is difficult to quantify how much of all information systems used daily is made possible by FOSS,

but considering that software such as Linux, OpenSSL, Apache and MySQL are all FOSS based it's clear how pervasive the use of FOSS is.

Carrier (2002) notes that digital forensics, in some basic incarnation, has existed as long as computers have. They continue by stating that in years gone by, digital forensics was a discipline limited to governments who used or developed proprietary tools to serve their needs. This has changed in recent years as the commercial adoption of digital forensics has led to the development of very competitive COTS as well as FOSS tools to aid digital forensics professionals. The lack of formal development procedures and documentation associated with FOSS meant that FOSS tools could be rapidly developed to satisfy needs of investigators as they arose, leading to the popularisation of ad-hoc tool development. Furthermore, the collaborative nature of FOSS and the fact that any person with the will and skill could contribute to the software, resulted in feature-rich toolsets being developed.

Both Carrier (2002) and Manson, Carlin, Ramos, Gyger, Kaufman, and Treichelt 2007 noted that of the issues with FOSS tools, ease of use is among the biggest. To a certain extent this is understandable, as most FOSS tools begin as purpose-built utilities to serve a very specific need, and the developer is also the primary audience and user.

The development of digital forensic tools is more important than ever before as existing tools become increasingly obsolete (Garfinkel, 2010) and as technologies, complex and proprietary data formats and protections like Full Disk Encryption (FDE) are more widely adopted. And, regardless of individual characteristics and adoption rates, both FOSS and COTS solutions have contributed and will continue to contribute to the advancement of the digital forensics discipline.

2.1.3 Challenges

Challenges to the practice of digital forensics are numerous and increasing. Due to the significant variance in tool functionality and build quality, a great many resources have been devoted to the validation of digital forensic tools. Validation efforts are extensive and justifiably so as a failure in a tool could potentially lead to the acquittal of the guilty (Gottehrer, 2016) or conversely, the condemnation of the innocent. One such effort that attempts to perform widespread validation of open source digital forensics tools is the Computer Forensic Tool Testing (CFTT) workgroup established by the National Institute of Standards in Technology (NIST). The CFTT aims, among other things, to baseline the performance and accuracy of a wide variety of tools against a standardised methodology

(Dykstra and Sherman, 2012). By doing so they hope to accredit the tools with the necessary level of trust to ensure utmost admissibility of the evidence they produce.

One of the issues facing CFTT as well as other validation frameworks is the lack of insight into known errors and failure rates for COTS and FOSS tools. Being able to establish a known error rate for a specific tool is important for the following reason: If a key piece of evidence can be based on a result of a tool with a known error rate of significance ¿5% or more, that evidence can be deemed unreliable and not admitted to court. If, however, evidence was produced by tool with a non-significant error rate in a testing environment, that evidence can be seen as forensically sound since the procedure is also known to be accurate.

This lack of transparency is part of the problem with tool validation and verification, especially with COTS. It is extremely difficult to establish a known error rate for a tool whose procedures are intentionally obscured (Carrier, 2002). It is understandable that the producers of COTS tools want to protect their intellectual property by not releasing source code or testing frameworks for the software, but this means that any users of COTS accept that exhaustive testing was performed on the software without much proof. There is, as noted by Carrier (2002), a commercial incentive for COTS providers to withhold important error metrics, which can result in users questioning the integrity of established software from a reputable vendor.

Carrier (2002) notes that there is a concerted effort to have digital forensic tools validated and verified to use their output as potential evidence in legal proceedings, and when it comes to FOSS tools there are two main issues to consider. Firstly, FOSS tools often lack any kind of formal testing as a consequence of the circumstances and environment they are developed under. Secondly, FOSS tools are easier to create validation tests for, since the design, process and code is open for anyone to review. Carrier (2002) also notes on this point that having open design standards and documentation allows testing methodologies to be developed with more ease and that the open nature of FOSS means that bugs and errors in the tool cannot be hidden. Carrier (2002) goes on to mention that FOSS tools can have a known error rate to justify confidence, since the defect history of the tool is in the public domain and verifiable. One of the reasons cited by Garfinkel (2010) for the decline of the “golden age of digital forensics” is the sheer variance in data and data formats used on all manner of digital devices today. It is not uncommon to see multiple different data storage and transmission protocols in a single standalone system, let alone a highly integrated and complex system. Garfinkel (2010) notes that it is increasingly common for issues to arise with data analysis due to format incompatibilities and other

similar factors.

Not only do digital forensic practitioners have to deal with increasingly complex data formats, but they also have to deal with the increase in the volume of data stored on devices. Garfinkel (2010) notes that, due to the storage capacity of modern digital devices, it can become impractical to perform basic tasks, such as creating forensic images of devices, in a reasonable amount of time.

Garfinkel (2010), Dykstra and Sherman (2012) and Dykstra and Sherman (2013) all noted that the recent and sustained adoption of cloud computing poses a major risk to traditional digital forensics methods as access to the data needed for an investigation does not necessarily reside with the party initiating the investigation. Part of the appeal of the cloud computing model is that it abstracts some of the technicalities of managing infrastructure or services from the end user and delegates that responsibility to the cloud provider. This means that even with mandate from the owner of a system, it can be troublesome to obtain the necessary log files to effectively perform an investigation. Dykstra and Sherman (2013) points out that in this model, a lot of trust is placed in third parties when it comes to validating the integrity of data; they event then developed a tool suite called FROST to address these issues of trust.

Apart from the technical challenges noted previously, there exist a host of challenges as well as opportunities regarding the legal aspects of digital forensics. Gottehrer (2016) notes the shift in the nature of evidence from primarily paper-based physical artefacts to digital mediums. They go on to note the implications this change might have on the legal fraternity and specifically noted that legal practitioners who do not understand the nature of digital evidence and forensics do so at their own peril.

The need for legal requirements in digital forensics investigations is well established, as noted by Kuntze, Rudolph, Alva, Endicott-Popovsky, Christiansen, and Kemmerich 2012, when they mention that incorporating legal views into device design can assist in maintaining the probative value of evidence produced by such devices. Although they specifically refer to devices, the same argument can be made for tools and software used during the investigation process. They go on to note that such efforts to ensure admissibility of evidence should be proactive, as any reactive effort would not add as much value.

When it comes to digital evidence produced by trusted devices, a healthy scepticism is still necessary. It must, as it were, follow the dictum that guides many other information security domains namely: trust, but verify. Specifically, Kuntze *et al.* (2012) notes the need to be able to verify that a digital record has not been modified.

One of the best known legal principles, as developed and implemented in the Unites States of America (US) court systems, pertaining to the admissibility of digital evidence is the Daubert standard. Kuntze *et al.* (2012) notes that the Daubert standard "...is often used to determine if scientific evidence, including digital evidence, is admissible as valid evidence in court."

Carrier (2002) notes that the Daubert standard can be used in US courts to determine the reliability of evidence presented during a trial. The Daubert standard usually applies to scientific evidence, or evidence of a technical nature that is not generally understood by judges or juries and was evidenced in its origin; Clarified by Carrier (2002) as stemming from the U.S. Supreme Court's ruling in Daubert vs. Merrell Pharmaceuticals (1993). The purpose of the standard is to verify the validity of scientific evidence, and by extension, make such evidence admissible in a court. The standard aims to verify the methodology as well as techniques employed to extract evidence and draw valid, true conclusions by asking a question on each of the following four topics:

1. Error Rate
    - Is there a known error rate for this procedure?
2. Testing
    - Has this procedure been tested?
3. Publication
    - Has the procedure been published and subjected to peer review?
4. Acceptance
    - Is the procedure generally accepted in the relevant scientific community?

Because of the nature and acceptance of the Dauber standard, it would be prudent to develop a digital forensic tool or process that can answer these questions easily and satisfy a court's demands for rigor, due process and validity.

2.1.4 Summary

Most efforts from acquisition to presentation are geared toward preserving the integrity of evidence and the chain of custody. Practices like hashing are a fundamental step in this preservation process and are often used as the only mechanism to prove integrity of evidence. Although the choice of hashing algorithm might change over time, the basic process has not and it is still performed in an isolated context with the practitioner or investigator being solely responsible for both creating the evidence signatures and verifying them.

Both FOSS and COTS tools play a huge role in almost all digital forensic practices and there is growing need to develop more advanced tools, as the evolution and adoption of technology threatens current digital forensic processes.

An important aspect of tool development is the question of testing and validation as the tools are very often used for reconstructing events and drawing conclusions from the data. Any tool output that is not verifiable and reproducible with a significant level of accuracy does not benefit an investigation.

The nature of digital forensics and its tight coupling with legal matters means that legal challenges often translate into substantial digital forensic challenges. As evidence from more and more connected devices are used in legal proceedings, the reliance on and accuracy of the data produced by these devices is more important than ever before. The use of data obtained from electronic devices to prove or disprove allegations in legal matters is not common practice and the rigor of legal scrutiny is now cast upon digital forensic methods, tools and practices. Legal concerns, like admissibility and digital forensics, will continue to develop and become ever more important as the two fields coexist.

Integrity lies at the core of the discipline of digital forensics. Many of the processes, tools and challenges depend on some level of verifiable integrity.

2.2 Blockchains

2.2.1 Introduction

Blockchain, the much-hyped (Lemieux, 2017) and often-misunderstood emergent technology, is set to redefine how complex problems of trust can be solved. For the purposes of

this work, blockchain will refer to the technology as a whole and not specific implementations of the technology. Where applicable, a specific implementation of a blockchain will be referenced accordingly. Blockchain is simply another form of applied cryptography, where existing cryptographic primitives like asymmetric cryptography, hashing, and Public Key Infrastructure (PKI) are combined to form a new technology that aims to solve a fundamental problem of trust.

What makes blockchain technology so appealing is the fact that it is distributed in nature, both in terms of trust and processing. The distributed nature means there is no single point of failure and a negligible possibility of undetected modification. As Lemieux (2017) succinctly noted: "Blockchains and distributed ledger technology promises trusted and immutable records in a wide variety of use cases involving recordkeeping, including real estate and healthcare". Lemieux (2017) further noted that that the main appeal of this technology is its ability to produce immutable and trusted records, foregoing the need for a trusted third party.

Blockchain technology is most well-known for its implementation in the form of the Bitcoin blockchain, a system that provides an immutable public ledger of transactions that can facilitate the transparent transfer of value between two parties without the need for a trusted third party or intermediary. Witte (2016) explains that this transfer of value can occur due to the transparent and unchangeable public ledger that forms the core of any blockchain system. This ledger provides both parties the assurance that the value being transferred belongs to a certain party and has not already been spent.

2.2.2 Purpose

The primary function of a blockchain, like the one in implemented in the Bitcoin network, is often unclear and misunderstood. To better understand the function of the blockchain a simple real-world scenario involving a bank and two bank customers can be used. to explain the principles at work.
Scenario: current banking transaction model
Patron A wants to transfer value, in the form of money, to Patron B. In the traditional banking model used by billions of people every day, Patron A would send a request to the bank to transfer an arbitrary amount of money to Patron B. As Patron A and Patron B both have accounts with the bank, the bank can facilitate the transaction by checking its ledger to see if Patron A has the necessary funds. If so, the bank deducts that amount from Patron A's account, updates the ledger to record the transaction and then adds that

amount to Patron B's account, again updating its ledger to record the increased balance in the account of Patron B. As simple as this seems, this type of transaction was only possible because both parties, Patrons A and B, trusted the bank as an intermediary to perform the transactions. Without this trust, this simple transaction would not have been possible as there would be no way for Patron B to be sure that Patron A has not already spent the money she needed to transfer. Similarly, there would be no way for the bank to ensure that Patron A does not spend that same money on a subsequent transaction. An updated and trusted ledger, maintained by the bank, is the only mechanism by which such a transaction can be concluded.

In the above example, the bank or intermediary can easily be replaced with the Bitcoin blockchain, because it is, fundamentally, a completely transparent ledger of transactions that records and displays all transfer of value in line with the key properties of blockchains (discussed later in Section 2.2.5.). In a traditional banking system, the bank or intermediary would be responsible for reconciling payments and balancing accounts, but this function is now completely and transparently facilitated in an automated fashion on the Bitcoin blockchain by willing participants. The main difference between a traditional banking ledger and the Bitcoin blockchain is that the Bitcoin blockchain is fully automated and there is no barrier to entry for potential participants.

Blockchains, such as the Bitcoin blockchain, have effectively solved the problem of counterparty risk and settlement risk by becoming a universally trusted ledger of transactions, controlled by a community of willing, incentivised participants, not a single entity. In terms of the risk mentioned, counterparty risk is the risk that either party to a transaction will not live up to their contractual obligation. Settlement risk, as noted by Peters and Panayi (2016) is: "the risk that one leg of the transaction may be completed but not the other". Through the mechanisms of immutability and transparency, a blockchain-type system can drastically reduce the need for a centralised third party, like a bank, to carry counterparty or settlement risk.

Cryptocurrencies, like Bitcoin, are a form of digital currency where the system for exchanging the currency is digital and the value of the currency is also digital. Although the Bitcoin blockchain is the most popular and most used of all such implementations, it is important to note that not all blockchains need manifest as cryptocurrencies. Blockchains can be used for a multitude of different applications, some of which are discussed later in more detail.

2.2.3 Design primitives

Witte (2016) notes that blockchain is based on two well-established cryptographic primitives: Public Key Encryption (PKE) or asymmetric cryptography and cryptographic hash functions. PKE is a very popular and universally used method of cryptography where a message is encrypted and decrypted with different, but related keys. This differs from more traditional, symmetric encryption where a message is encrypted and decrypted with the same key. Cryptographic hashes, as already discussed, are the output of a deterministic function that takes input of variable length (pre-image) and produces a unique value of fixed length (hash) for that specific input. Hashes are computationally infeasible to reverse to determine the pre-image of a given a hash.

Readers who are interested in the details of how these properties arise are referred to an authoritative book such as Schneier (1993). For the purposes of this work, it is sufficient to understand that cryptographic hashes have, among other, the following two properties:

- collision resistance, and
- pre-image resistance

Because a hash function produces a fixed length output regardless of the length of the input, there exists the possibility that two different input values will produce the same hash: a hash collision. Some hash functions, with a shorter length output and subsequently, lower entropy, are more prone to collision than others. Cryptographic hashes that are less likely to suffer from such collisions are said to be more collisions resistant. Without collision resistance it would be trivial to create identical hashes for different pre-images, negating the value of cryptographic hashes as a mechanism to verify the integrity of data.

Similarly, hashes have different degrees of pre-image resistance. Pre-image resistance is a measure of the difficulty to extrapolate the pre-image (input value) of a hash, given only the hash function output. To perform a pre-image attack to determine what input produces a given hash it would be required to enumerate all possible inputs and compare the output of the hash function to the hash of the unknown data - if it matches, the data provided as input to the function may be the same as the original unknown data. A lack of pre-image resistance could result in weak confidentiality as the pre-image of a hash of sensitive data, like a credit card number, can be easily computed. There is a reliance on pre-image resistance to preserve the confidential of data that has been hashed.

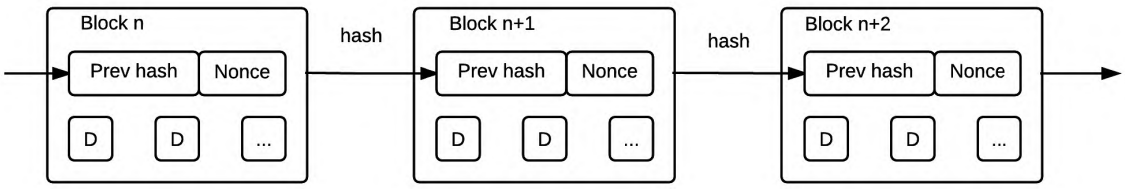


Figure 2.1: A visual illustration of a blockchain

Combining these two basic concepts, PKE and hashing, Nakamoto (2008) proposed the Bitcoin blockchain upon which all subsequent blockchain implementations to date, are based. Figure 2.1 is a simplified visual representation of a blockchain-type system, where every block is dependent on the contents of the previous block.

In Figure 2.1, 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, as explained by Rogaway (2004) in the context of cryptography, is simply a value used once for a particular message or operation. Schneier (1993) further noted that a nonce is usually a random value. By chaining blocks together like this, it is possible to verify the data in it, as any change in the data will result in a change of the hash which will necessarily cascade down the chain, changing all subsequent block hash values.

Merkle Hash Trees (MHT)

Introduced to modern cryptography and computer science by Merkle (1980) in the early 1980s, MHTs are another cryptographic primitive that makes blockchain technology possible and practical. The initial use case for MHT, as is clear from Merkle (1980) and the associated patent filing, was to facilitate the proof of a digital signature for the purpose of authentication. Again, the use case in blockchain technology is slightly different but rooted in the same principles.

MHTs rely heavily on 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 services and protocols to facilitate the validation of data

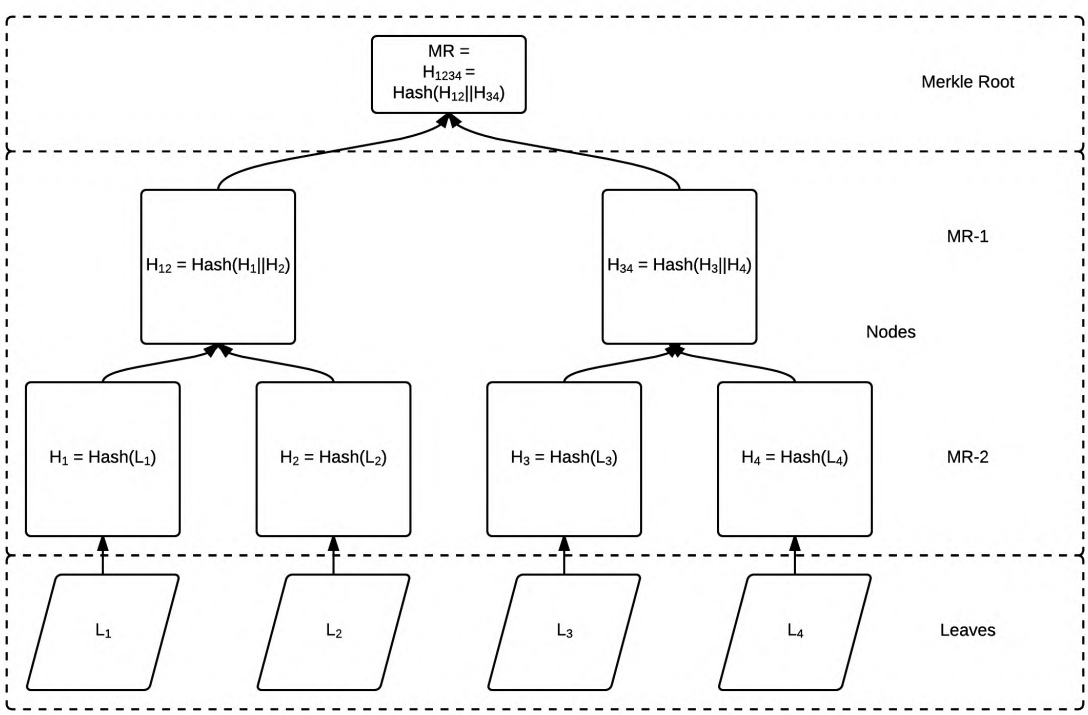


Figure 2.2: A symmetric binary Merkle hash tree

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 or this exact same purpose.

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

Figure 2.2 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 on the tree. One level up (level MR-2) there 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.

Figure 2.2 is an example of a binary MHT, I.E., a tree where each node has at most two children. MHTs can also be non-binary where each node can have more than two children. The MHTs that are used to validate transactions in the Bitcoin blockchain are binary trees.

Binary MHTs can be asymmetric, meaning that there is not an equal number of leaves as can be seen in Figure 2.3. Due to the nature of binary MHTs, the computation to get to the root is slightly adjusted where necessary, by duplicating the leaf or node to fill the spot of the missing node. By following this basic rule, computation on the tree are predictable and standardised.

Binary MHTs are only valuable if operations can be performed on them, which is why the list of possible operations supported by binary MHTs include:

1. Search
2. Insert
3. Delete
4. Traverse

Calculating the MR is also sometimes referred to as ‘collapsing the tree’ as it reduces the various leaves into a single root hash value.

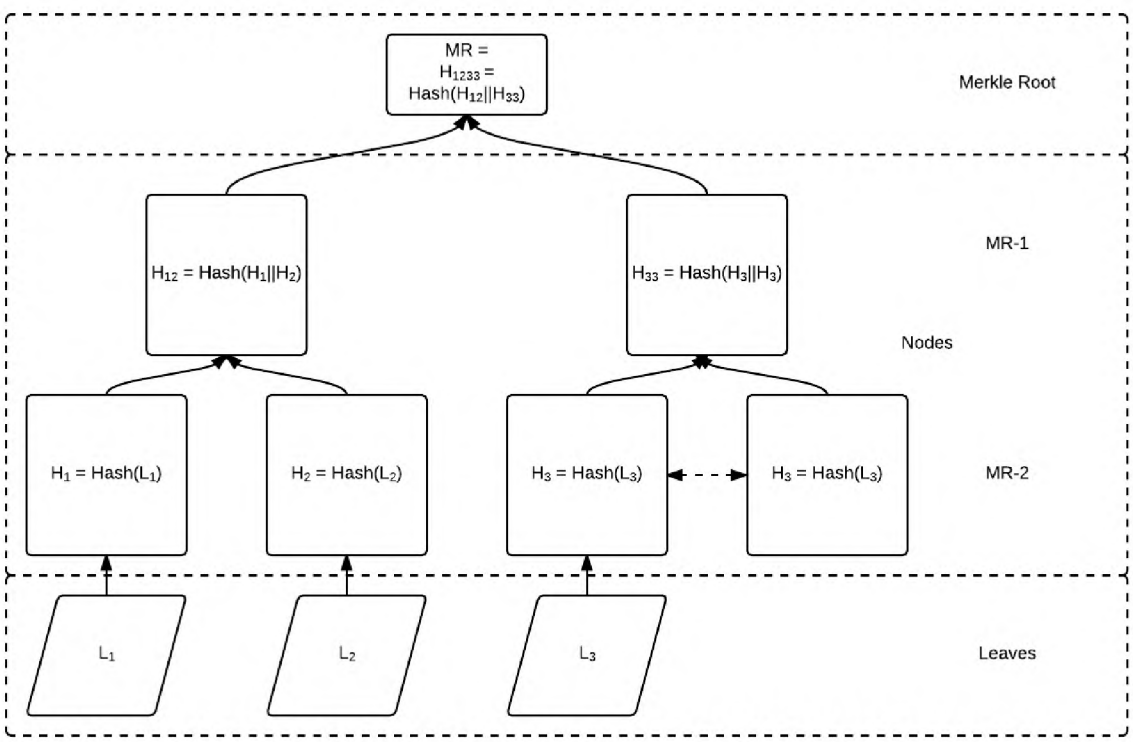


Figure 2.3: An asymmetric binary Merkle hash tree

Table 2.1: Merkle tree operation complexity

	Average	Worst
Search	$O(log_2(n))$	$O(log_k(n))$
Traversal	[1] $O(n)$	[1] $O(n)$
Insert	$O(log_2(n))$	$O(log_k(n))$
Delete	$O(log_2(n))$	$O(log_k(n))$

Due to the nature of MHTs, the performance of these operations varies, this is very important as quick lookups are essential to the use of MHT in most applications like Bitcoin. Brilliant.org (2015) gives an overview of the complexity (Big O notation) of certain operations on a MHT as seen in Table 2.1, where the branching factor (number of children of each node) is denoted by $_k$ for non-binary trees with n number of nodes.

A trees grow in size and complexity, so too does the complexity of operations on that tree. 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 MHT preserve space and data

[1] Amortised value estimate - averages out the worst operations over time.

Table 2.2: 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	*1-9 bytes*	Transaction count in vtx[ ]
	vtx[ ]	Transaction	*Variable Array*	Array of transactions

transfer operations.

2.2.4 Case study: Bitcoin blockchain

Bitcoin, the first implementation of a blockchain-driven system, serves as an appropriate reference implementation to better explain general blockchain functionality. Although not all blockchains follow this exact model, they are all based on the same basic principles.

Blocks

Bitcoin blocks are collections of structured data that form a fundamental part of the ledger. A block can be separated into two main components: a header and a body (also called the payload). The head of a block contains some reference data, including the block structure version and a reference to the previous block, while the payload of a block contains transaction data. The exact structure of a block can be seen in Table 2.2.

nVersion, one of the more notable fields in this discussion, is an indication of the block structure version. This field is necessary as the block structure might have changed over time and block parsing systems will need to know the version of the block to ensure compatibility.

HashPrevBlock is, as the name indicates, a hash of the header of the preceding block. The hash is a double-SHA256 hash function (dSHA256) of the concatenated content of the previous block header:

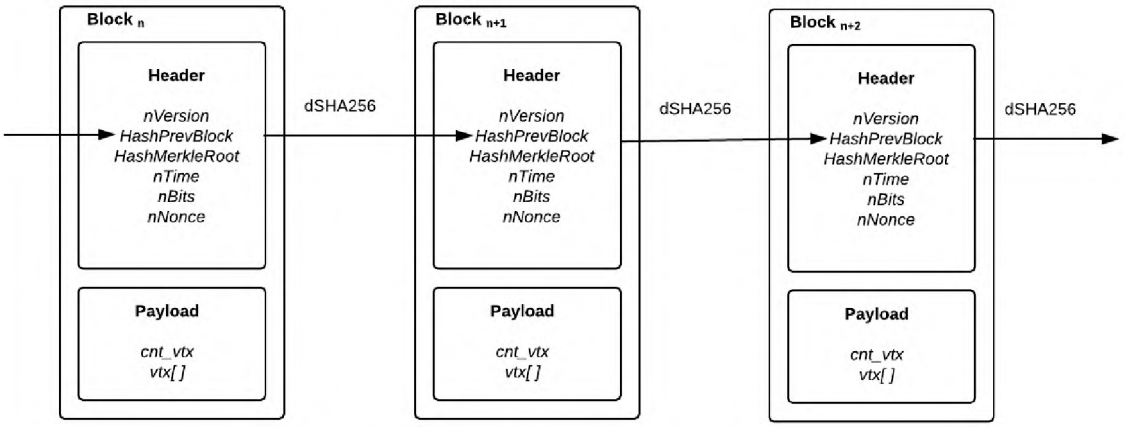


Figure 2.4: Linking of blocks in the chain through *HashPrevBlock* value

$HashPrevBlock_n$ = dSHA256($BlockHeader_{n-1}$) = dSHA256($nVersion_{n-1}$‖$HashPrevBlock_{n-1}$ ‖$HashMerkleRoot_{n-1}$‖$nTime_{n-1}$‖$nBits_{n-1}$‖$nNonce_{n-1}$)

It is this process, the hash referencing the header of the previous block, that facilitates the formation of align chain. By having a *HashMerkleRoot* in each block, apart from the very first block, a chain is formed that indicates in which order blocks were incorporated into the chain, as can be seen in Figure 2.4.

HashMerkleRoot is the root of the MHT computed over all the transactions included in *vtx[ ]*, using dSHA256.

nTime is the time in UNIX format of the creation of the block in question.

nBits stores the target value, denoted as T, used in Proof of Work (PoW) calculation.

nNonce is another component of the PoW puzzle, and is a simple nonce that can be used during the PoW calculation as a source of randomness to get to reach the target T. *nNonce* is explained in more detail in the section on PoW.

vtx[ ] is an array of transactions. Transactions are themselves a complex data type comprising of a range of other properties.

cnt_vtx is the total number of transactions included in the block in the *vtx[ ]* array.

At the time of writing (August 2017), a Bitcoin block can be up 1 024 kilobytes (1 024 000 bytes) in size, but no larger. Blocks, larger than 1 024 000 bytes, are considered invalid and will not be accepted by the network. As can be seen from the size allocations

Table 2.3: 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*	Reference to 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

in Table 2.2, the header data for a block (all non-transaction data) can be up to 80 bytes in size leaving the vast majority (1 023 920 bytes) for transaction data.

Transactions

Bitcoin transactions are collections of reference data; inputs and outputs specifying, amongst other things, the source and destination of the transaction as well as the value of the transaction. A regular Bitcoin transaction structure can be seen in Table 2.3

nVersion, as in the case of a block, indicates the version of the transaction structure (which may change over time).

Another notable field in the transaction structure is *scriptSig* that specifies the conditions under which the transaction output can be spent. The script, Turing complete scripting language built into the Bitcoin protocol, is called Script.

Similarly, *scriptPubKey* is another instance of Script that specifies the conditions under which the output of the transaction can be claimed. These fields are primarily used for embedding data as they are completely under the control of the user and can, compared to other fields, store relatively large amounts of data, according to Okupski (2015).

Bitcoin addresses are unique alphanumeric strings of characters that are used to identify the source or destination of a Bitcoin transaction. These addresses can be 27-34 characters in length, and are constructed through deterministic function with a variety of inputs. The reason for the varied length of addresses is as a result of the final address being encoded as a *base58* reduced character set. *Base58* is used as the encoding of choice since it removes some of the ambiguous characters from the *base64* character set like 0 (zero) and O (uppercase o), to avoid ambiguity in addresses and thus instances of erroneous transfers. At the time of writing two types of Bitcoin addresses exist: Pay-to-PubkeyHash (*P2PKH*) and Pay-to-ScriptHash (*P2SH*).

In the case of *P2PKH*, the basis of the address is the hash of the public key portion; the public-private, Elliptic Curve Digital Signature Algorithm (ECDSA) keypair, associated with a user or their wallet, a secure digital storage mechanism for Bitcoin. The alphanumeric address is constructed by computing the *SHA256* and *RIPEMD160* hash of the public key to produce the *pubKeyHash*. A version byte is then prepended to *pubKeyHash*. Once the version byte has been appended, a checksum is calculated over the concatenation of *pubKeyHash* and the version byte by performing a dSHA256 hash over it and truncating the result to the first four bytes. Finally, the checksum is appended to *pubKeyHash* and the result is encoded using the *base58* character set resulting in the final *P2PKH* address.

The process for *P2SH* addresses is similar but uses the value of the redemption script located in the *scriptPubkey* field as the initial input rather than the hash of the public key, as in *P2PKH*. The process from that point onwards is exactly as detailed for *P2PKH* and results in the final *base58*-encoded *P2SH* address.

Aside from the two Bitcoin addresses used, the Bitcoin protocol also defines coinbase transactions. These transactions do not facilitate the transfer of value between participants in the network; They are exclusively used to carry transaction fees that are rewarded to nodes for processing regular transactions.

Chain

The chain as elaborated on earlier in the explanation on blockchain functionality, 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. Figure 2.5 is a high level view of blocks in the Bitcoin blockchain.

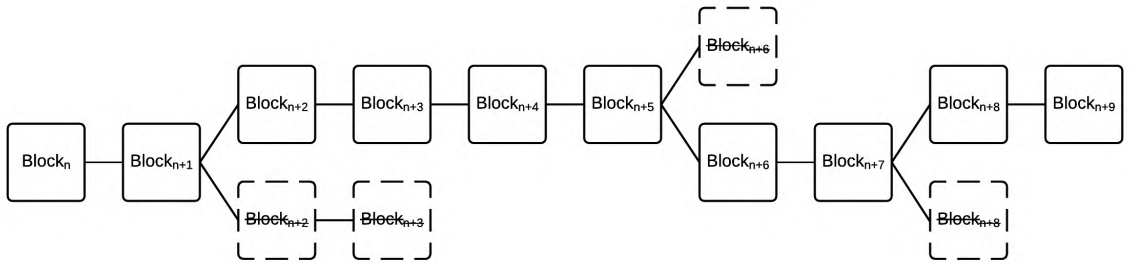


Figure 2.5: Example of a Bitcoin blockchain with forks

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 (as defined in Table 2.2), and then
3. then solve a PoW puzzle associated with that block.

Once this puzzle is solved, the miner broadcasts the proof along with the block, and other miners then proceed to verify that proof. If the proof is accepted, the block is added to the chain as the most recent block and the miner is rewarded for the work it has completed. Once a new block is added to the chain, the hash of its header is used by the miners in the network to create a new block, add waiting transactions to that block structure, and repeat the whole process.

This ongoing work results in the chain as depicted in Figure 2.5. 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 Figure 2.5 as the block with dotted borders. However, transactions that were part of these orphaned blocks are not lost but as instead rebroadcast to the network for inclusion in the next block. 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.

As discussed, a fork in the blockchain can occur naturally as a matter of chance or it can be an induced fork. These occur when the majority of nodes agree to reprocess a previous block to create an alternative chain, invalidating the revious chain. Induced forks are not a common occurrence but are usually the subject of controversy in the community of Bitcoin nodes (Redman, 2017).

Controlled supply and incentive

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 reward is given to the miners that solve the PoW puzzle, measured in units of Bitcoin currency (BTC), is how new Bitcoins are introduced into the system. Rewards are variable and diminishing geometrically over time to ensure that the rate at which Bitcoins are created remains constant as the overall mining power of the network increases. This diminishing reward for finding a block is called the block reward halving, and occurs every 210 000 blocks. This means that after every 210 000 blocks, the reward for finding a block is reduced by 50%. In the initial Bitcoin blocks, the reward was 50BTC. At the time of writing, the reward halving has happened twice and the block reward has reduced to 12.5BTC. By strictly applying this reward halving, the creation of new Bitcoins will effectively stop after 64 halving operations as the reward for finding a block will be less than the smallest unit of Bitcoin, a Satoshi. Estimates on when this point will be reached vary, as it's based on factors such as mining power and technological advances; some sources, however, estimate the year 2140 (Bitcoinwiki, 2014).

Proof of Work

PoW 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 ag-

gregate 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 (2008) was based on the hashcash system developed by Back (2002). 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 target T, the nBits field, in Table 2.2, in a block header.

To solve a PoW puzzle, a miner must calculate a dSHA256 hash H over the contents of the block so that it is smaller or equal to the target hash T such that:

$H \leq T$

As discussed before, hash functions are deterministic, and input i will always result in the same output o when passed through the same function dSHA256.

$H = \text{dSHA256}(i) = o$

In order for a miner to generate variable hashes to satisfy the condition $H \leq T$ over the static contents of a block i, it needs to incorporate other random data n in the form of a controllable nonce into the input of the function so that:

$H = \text{dSHA256}(i \parallel n) = o_n$

Since the miner cannot change the contents of i, it varies the input of n, concatenates i and n, performs the hash function and evaluates the result against the condition $H \leq T$. If this 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 requiring that value T starts with a certain predetermined amount of zeros. The more zeroes required, the more hashing operations the miner has to perform in order to find a value of H that satisfies the condition $H \leq T$.

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.

2.2.5 Key properties

Blockchain technology, as known and implemented today, is only useful because of a few key properties which make the concept and practical execution possible, namely:

1. Immutability,
2. Chronology,
3. Redundancy, and
4. 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. Witte (2016) highlights that this immutability is conditional and strengthens over time as a consequence of the design of the system. 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. According to Witte (2016), in the Bitcoin blockchain, the current block depth to guarantee a permanent and unchangeable transaction is six-blocks deep. 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 in the public ledger, all information can be considered secure, unchangeable and a true record of data and transactions over time.

Chronology - the sequential arrangement of events or transactions over time - is a another property of blockchain design that gives it immense value and utility. The necessity for this property is immediately visible when considering the current application of the technology (I.E., an accurate representation of a transaction position relative to other transactions, in time) is essential for a ledger. Without the accurate timestamping of transactions in a blockchain, it would be infeasible to determine when a transaction or event occurred. Timestamping is the ability to associate the existence of a certain piece of information with a specific moment in time, according to Gipp, Meuschke, and Gernandt 2015. As noted by Nakamoto (2008), the timestamping of blocks is one of the first steps in creating the block. Gipp *et al.* (2015) also notes that protocols for decentralised timestamping can significantly increase the overall security of such timestamps and the trust that can

be placed in them. 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 and resiliency.

Redundancy is a further significant property of a blockchain-based system, and was a key design consideration in the design presented by Nakamoto (2008). This was especially pronounced when Nakamoto (2008) realized that 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. For a blockchain system to work effectively, there needs to be a significant number of participants that contribute to the growth of the chain, and for information processing to be embedded in the chain. Decentralised trust, or the lack of trust in a single entity, implies that trust is the responsibility of the system and not that of a governing entity or a subset of privileged entities. Furthermore, for entities to participate in this ongoing hash-based PoW, they need to maintain a copy of the updated ledger, which means that there are multiple copies of the true data source, and that the failure of one or a few nodes would not significantly affect the resiliency of the system. As Gipp *et al.* (2015) highlighted, the decentralised design could lead to difficulties if entities are not incentivised to participate. Nakamoto (2008) knew this would be the case and therefore embedded an incentive system into the core design of the Bitcoin blockchain. By constantly incentivising entities to continue working on processing and creating the chain, Nakamoto (2008) ensured the continued survival of the system. Most blockchain-based systems have incentive systems and each differ slightly in terms of the frequency and value they reward. Not only is the incentive ingenious in the way it keeps the distributed nature alive, it even caters for and adjusts to environmental aspects to keep the incentives fair and relevant. By having a completely distributed system with decentralised trust, the resiliency of the system can be guaranteed for as long as there is an attractive incentive to participate in the system.

Transparency is the final the four core blockchain properties and is more of a functional requirement and not a design consequence. Considering the prevailing application of blockchain-based systems similar to Bitcoin, it is obvious that for the system to work, all transactions need to be broadcast openly to any entity willing to listen. Apart from the broadcasting of transactions, the information embedded inside the ledger should also be open for all to see and verify. Transparency is fundamentally necessary for the system to function and it cannot be changed. In the case of the Bitcoin blockchain, no balances are stored, only transactions. So, in order to calculate the balance of a specific address, all the transactions in and out of that address need to be visible. Transparency of the technology and the information processed by the system also increases trust in the system.

Malkovský (2015) notes that not only is it beneficial having the data in the blockchain transparent, but the transparency of the protocol also enables trust in the system.

By combining immutability, chronology, redundancy, and transparency, blockchain-based system are uniquely equipped to address many of the problems associated with trust and decentralised processing. The general hype around this technology is by no means unfounded, but it is slightly inflated and misunderstood. Like any technology, blockchain technology has both strengths and weaknesses, and it is important to understand these weaknesses to understand which types of problems blockchain technology can and cannot solve.

2.2.6 Weaknesses

Blockchain technology is not a universal solution to all technology problems. Even though there is a plethora of potential applications for the technology, not all of them are good or even practical. Below is an overview of why blockchain technology cannot solve all problems and what some of its weaknesses are.

The first and most notable weakness in a blockchain based system is rooted, quite ironically, in trust, the very problem it tries to solve. The weakness comes in the form of a theoretical attack, called the 51% Attack. Witte (2016) notes that the number, 51, in the name has little relevance to an actual attack, but rather serves to illustrate the nature of the problem. The 51% Attack would occur if a majority of participating nodes in a blockchain system colluded to manipulate the addition of new blocks to the chain. Because the system is designed to achieve consensus through computational power, if a majority of nodes with a significant portion of the entire network's computational power colluded, they could theoretically have a small time window to manipulate recent blocks to their advantage. The attack is theoretical and mitigated through the concept of deeper block depth I.E., as subsequent blocks are added onto the block it becomes more and more impenetrable. Witte (2016) notes that the 51% Attack creates a problem of speed rather than security, as users must wait for a block to be embedded deeper in the chain before fully trusting its contents.

Speed is another potential weakness. As alluded to before, for a block to be considered safe it needs to be a few levels deep in the chain and this processing takes time. The exact amount of time is dependent on design factors in the blockchain implementation. For example, the average block release time differs significantly between Bitcoin and Litecoin

according to Volf (2016), with Bitcoin at 10 minutes and Litecoin at 2.5 minutes. Litecoin will be discussed in more detail later, but in short it is a Bitcoin spinoff with the specific aim of shorter transaction confirmation times. Litecoin aims to achieve this speed by releasing blocks more often that Bitcoin does by broadcasting the necessary block data to all participating nodes, enabling them to start mining that block. Speed is not a problem that is easily addressed in blockchain-based systems as their fundamental design necessitates a certain level of computational effort to establish the distributed consensus through mechanisms such as PoW. Blockchain system speed has also been the topic of much prior research, as can be seen in Kiayias and Panagiotakos (2015) where they noted the conflicting relationship between PoW (security) and speed.

Directly related to the issue of speed is the issue of size, or rather the lack thereof. In Nakamoto's original Bitcoin blockchain design there was some consideration about minimising the size of the blockchain, since a complete copy would need to be stored by each node in the network. It is also the case that due to its nature, a blockchain could only ever get bigger and never have information removed from it. The challenge of space has meant that significant work has been done on best utilising the available space on a blockchain, notably by Okupski (2015) in which the available space on the Bitcoin blockchain for embedding arbitrary data storage was analysed and maximised through novel techniques. The purpose of this was to utilize the immutable nature of the Bitcoin blockchain to facilitate the development of an anti-censorship tool as a way to embed messages on the blockchain, making them impossible to alter or remove. The distributed nature of a blockchain-based system requires that the data exists as redundant copies, and for that to be true, there needs to be a focus on the frugal use of space in these systems. It is important to note that there is no theoretical size limit on the size of the entire blockchain, but rather the size of individual blocks. Theoretically, the blockchain has the potential to store an infinite amount of data but that data would have to be spread over an infinite number of individual blocks of finite size. This finite size of individual blocks gives rise to an impracticality when using a blockchain simply as a mass storage mechanism.

2.2.7 Popular blockchain applications

Bitcoin, as discussed, is the first and most popular implementation of a practical blockchain-based system. Even though the Bitcoin blockchain initiated the concept, it is no longer the only blockchain system. The explosive growth and popularity of the blockchain concept gave rise to a complete ecosystem of alternative blockchain implementations. Some of

these alternative digital currencies attempt to solve the shortcomings of Bitcoin, whereas others diverge completely from the digital currency use case for blockchains. To date there have been numerous implementations and usages of blockchain technology as is obvious from titles like: "101 TOP BLOCKCHAIN COMPANIES" in Rampton (2016). The following is a short overview of some of the major blockchain systems, apart from Bitcoin, that have driven the technology and its adoption to new heights.

One of the first successful blockchain implementations other than Bitcoin was an alternative to Bitcoin. Litecoin was and remains a very close copy of Bitcoin, both in terms of technology and purpose, with only a few subtle differences. Litecoin was created to address some of the practical issues of Bitcoin, primarily the slow transaction times. The Litecoin website at Litecoin Project (2017), describes Litecoin as: "...a peer-to-peer Internet currency that enables instant, near-zero cost payments ... Mathematics secures the network and empowers individuals to control their own finances. Litecoin features faster transaction confirmation times and improved storage efficiency than the leading math-based currency. With substantial industry support, trade volume and liquidity, Litecoin is a proven medium of commerce complementary to Bitcoin". Litecoin, despite its faster transaction confirmation and reduced size, did not make Bitcoin obsolete and although still in existence, has fallen in popularity due to a decreasing market cap. Despite this, Litecoin is significant as it was the first blockchain implementation that attempted, and arguably succeeded, in addressing some of the issues with Nakamoto's initial design. To be clear, Litecoin built on the basic principles Nakamoto posited, but it improves speed by tweaking certain parameters, like the time between block creation and the overall size of blocks. By releasing larger blocks more frequently, Litecoin is able to process more transactions per second than Bitcoin.

Another notable alternative cryptocurrency is Zcash. According to the developer, (Zerocoin Electric Coin Company, 2016): "Zcash is the first open, permission-less cryptocurrency that can fully protect the privacy of transactions using zero-knowledge cryptography". As with Litecoin, Zcash is a blockchain implementation that attempts to solve a problem identified in Bitcoin, namely, privacy. As mentioned before, in the Bitcoin blockchain all transactions are inherently public and anonymity is only guaranteed to the extent that your Bitcoin address is not tied to your identity. As for transaction sources, destinations, and values, those are all public in the Bitcoin blockchain. Zcash addresses this privacy concern by making use of a cryptographic tool known as Zero Knowledge Proofs (ZKP), which allows the transfer of value anonymously while still ensuring the integrity of the public ledger on which it relies. In the Zcash implementation, the ZKP construction is called a zero-knowledge Succinct Non-interactive Argument of Knowledge

(zk-SNARK). Not only does Zcash allow shielded transactions - transactions of which the content is encrypted - but it also allows normal public payments like the Bitcoin blockchain. Zcash is a promising and new addition to the cryptocurrency ecosystem and illustrates how blockchain technology might be improved upon.

Among the most notable blockchain technologies that departs from the most common use case of such technology is Ethereum. As described by Ethereum Foundation (2016): "Ethereum is a decentralized platform that runs smart contracts: applications that run exactly as programmed without any possibility of downtime, censorship, fraud or third party interference". Or, more concisely, Ethereum is a generalised platform for computing based on the blockchain concept. Wood (2014) notes: "Ethereum is a project which attempts to build the generalised technology; technology on which all transaction based state machine concepts may be built. Moreover, it aims to provide to the end-developer a tightly integrated end-to-end system for building software on a hitherto unexplored compute paradigm in the mainstream: a trustful object messaging compute framework.". Whereas Bitcoin, Litecoin, Zcash utilise blockchain technology for a very specific use case, Ethereum is abstracted and presents a platform for implementing solutions for a vast array of different use cases. One of its most notable use cases is the concept of smart contracts. These smart contracts are written and embedded into the Ethereum blockchain, ensuring the terms of the contract are immutable and the execution, given the right circumstances, is guaranteed.

Using Ethereum and its powerful, built-in Turing-complete, scripting capabilities, entities such as the Distributed Autonomous Organization (DAO) were brought into being. According to del Castillo (2016), the DAO is a distributed, leaderless organisation, built on top of the Ethereum blockchain. Its purpose is to serve as a vehicle for supporting Ethereum-related projects. A participant in the DAO can be seen as a stock holder in a traditional organisation, and gets to exercise their vote on which projects should be funded. In fact, the DAO itself was built on a set of smart contracts that are embedded in the Ethereum blockchain. According to Delmolino, Arnett, Kosba, Miller, and Shi 2016, smart contracts are user defined programs that stipulate a set of rules to govern transactions that are enforced by a network of peers.

Although the DAO remains popular, it has suffered a number of setbacks, one of which was a hack that threatened to invalidate the whole system and the principle it was based upon. As Finley (2016) notes, the DAO was the victim of an attack that exploited an error built into the smart contract governing the DAO. The attack was able to drain large sums of digital currency from the DAO. Not only was the attack very effective, it

was also a great point of contention as the purveyors of the DAO performed a fork of the blockchain in order to reclaim some of the lost funds. A blockchain fork (or forking) refers to the action of choosing a point on the blockchain, prior to some unwanted action, and then creating an alternative chain; the chain is processed from that point onward. Forking a blockchain is a non-trivial task and involves consensus from a majority of the nodes processing the blockchain. By forking a chain, the community of nodes effectively has the power to 'go back in time' undoing some unwanted action and processing ahead as if that action never occurred; creating an alternative timeline. It is important to note that the act of forking the chain does not erase the record of the unwanted action, but rather works around it by creating an alternative chain. This was very contentions since the DAO was built on the principle that the smart contract is the only law that matters, but when it was found that the smart contract was flawed, that stance was very invalidated in order to save the DAO members from losing money. As a headlines stated: "A $50 Million Hack Just Showed That the DAO Was All Too Human" (Finley, 2016). Because of the versatility of the Ethereum platform, there are many other systems and applications built on top of it and even more being planned.

Despite the seemingly devastating effects and monetary loss associated with the DAO hack, the fundamental security of blockchain technology was never compromised. The DAO hack took advantage of a logic implementation error in a system built on top of the Ethereum blockchain. It did, however, illustrate the immutable nature of the blockchain and how a hard fork was necessary to recover funds. Even after such a hard fork, the offending transactions were not deleted but remain in an immutable state in an abandoned fork of the Ethereum blockchain for all to see.

2.2.8 The future of blockchains

The understanding of blockchain technology, its strengths, weaknesses, and possible applications is slowly increasing (Marc, 2016) in technological and non-technological circles. Although it is clear that the application of blockchain technology stretches far beyond digital currency, financial applications are currently the main driver for the adoption of the technology. The financial sector worldwide have taken note of blockchain technology, with many large financial institutions, such as the Bank of England, actively investing, researching and developing the technology (Barrdear and Kumhof, 2016). Blockchain has also not gone unnoticed by governments around the globe (Curry, 2016).

As noted by Filippi (2013), Bitcoin, and blockchains in general, have been the subject

of various legislative attempts in the EU. Filippi (2013) also notes that the development of such legislation is driven primarily by the threat cryptocurrencies pose to financial systems and governments of the world. As a measure of the impact of the technology and its applications, legislative intervention is a very convincing metric. A recent bill, noted by Arizona State Legislature, Fifty-third Legislature, First Regular Session (2017) was passed in the state of Arizona in the United States of America, that gives legally binding status to smart contracts and blockchain signatures. This is a significant step forward in the future adoption and acceptance of blockchain-based technologies.

2.3 Digital forensics and blockchains

2.3.1 Current state of research

Both digital forensics and blockchains are concepts that are fairly new to modern computer science, and have had the bulk of their development and innovation happen after the turn of the century. In fact, as discussed, blockchain technology only really came into being around 2008 and only recently started drawing widespread attention. The technology behind digital forensics and blockchains are heavily rooted in modern cryptography, though they pursue very different goals. What the underlying cryptography provides, to both, is a mathematical instrument to provide reasonable assurances of integrity and trust.

Seeing trust and integrity as a common base for many of the functions within digital forensics and blockchain technology, enables the appreciation of how these two seemingly unrelated concepts can be married. It also provides perspective and the ability to develop use cases where the properties of blockchain stand to benefit digital forensics as well as other areas of computer science.

Having discussed blockchain technology and digital forensics, exploring some of the works that demonstrate this merger of concepts will provide further understanding of the versatility of blockchain technology. Apropos to that is a discussion of previous work that, to varying degrees, explores the application of the trust and integrity, provided by blockchain, in digital forensics and other fields.

2.3.2 (Ab)using Bitcoin for an anti-Censorship tool - Okupski (2015)

This work centres on using the trust, integrity, resilience and immutability provided by the Bitcoin blockchain to augment anti-censorship tools.

Okupski (2015) notes that freedom of speech is a very important and often undermined basic right, and certain corporations and governments actively blocking individuals from exercising this right. Okupski (2015) specifically references the censorship activities of the People's Republic of China when describing these injustices toward free speech.

Okupski (2015) goes on to present Bitcoin and the blockchain as a possible mechanism to circumvent this censorship and suppression of free speech. At first, it might seem odd to make this suggestion, but there are two features of Bitcoin that make it ideally suited to such applications. Firstly, the political and economic nature of Bitcoin makes it incredibly difficult for lawmakers to enforce laws on. Bitcoin is also difficult to fit it into any existing legal framework, making it difficult to enforce outright bans. Secondly, Bitcoin and the infrastructure on which is it based, is truly global and decentralised. The pseudonymous nature of participation and decentralisation of infrastructure would make it very difficult for oppressive regimes to enforce restrictions upon it.

Okupski (2015) also establishes that the act of embedding arbitrary data into the Bitcoin blockchain is not new and has been in practice as early as the genesis block in the Bitcoin blockchain. Data embedded in the blockchain includes various seemingly random messages, images and quotes from political figures such as Nelson Mandela and even a script that allows users to embed data in the Bitcoin blockchain.

After establishing that embedding data into the Bitcoin blockchain is possible, Okupski (2015) continues to describe, in detail, the Bitcoin protocol to identify elements that can be used for embedding data and thereby improving the efficiency of current techniques. They find that current methods are functional but not very efficient and that by using improvements in the evolving Bitcoin protocol these methods can be improved. Currently, the data is embedded in the destination address field of a transaction as this parameter is under the control of the user performing the transaction. A consequence of using this method is that the Bitcoin value associated with that transaction is lost forever (a financial cost) as there is a mathematically insignificant chance of the arbitrary data, now used as the destination address, would correspond to a valid Bitcoin address, perhaps even an address belonging to the sender. Additionally, as there is limited space available per

transaction, multiple transactions are needed to embed data of significant size - this in turn could have a significant cost implication due to the un-spendable transaction outputs. This places two restrictions on the process of embedding data: the cost and space.

To counter this, Okupski (2015) developed a new, more efficient method for embedding data, thereby significantly reducing the cost per byte of data. They found that by using the Pay-to-ScriptHash (P2SH) transaction type together with other methods, they could reduce the cost of embedding data to around 16 Satoshi (the smallest denomination of Bitcoin) per embedded byte of data. This translated to approximately $184.21 per Megabyte of embedded data, given the Bitcoin price of $1097.98 per Bitcoin at the time of writing.

The result of this research is an application that has the ability to efficiently encode and embed arbitrary data into the Bitcoin blockchain. Okupski (2015) has also released the code for this application publicly in order to further the cause of anti-censorship movements as well as promote future research on the topic.

Unfortunately, although this method significantly improves the efficiency and, by extension, reduces the cost of embedding data into the Bitcoin blockchain, it can still not be considered inexpensive. It is, after all, subject to Bitcoin price fluctuations and may become cheaper or more expensive as time goes on. Even with the improved efficiency, this method does not seem like a viable solution to embed large amounts of data in the Bitcoin blockchain.

The work done by Okupski (2015) is, however, very informative and showcases how blockchains, in this case the Bitcoin blockchain, can be used for purposes other than what it was initially intended, despite it carrying a cost. Okupski (2015) goes on to highlight that by embedding data into the blockchain users can prevent the data's destruction and ensure its propagation to all corners of the world. It is clear that Okupski (2015) realised the advantages of the four core blockchain properties - immutability, transparency, chronology and redundancy - and used them to create a use case completely out of the scope of the initial Bitcoin design.

Even though the work in question does not directly refer to any digital forensic practices, the nature of the problem it is trying to solve resonates with digital forensics in that data is persisted and protected from tampering. In the use case by Okupski (2015), the data that is persisted might be messages and important political statements, but in the case of digital forensics the data could very well be evidence or signatures of evidence.

2.3.3 Bots, block chains and believable logs - Weilbach (2014)

This work, also by the author of this paper, introduces the idea of marrying the fields of digital forensics and blockchain, and informs this dissertation, which itself is addressing some of the potential applications noted in Weilbach (2014).

Weilbach (2014) starts by introducing the concept of digital forensic readiness and Linking that to technologies such as Intrusion Detection Systems (IDS). He then goes on to highlight the necessity to have legally admissible evidence to facilitate prosecution and further the cause of digital forensics in general. In this work, Weilbach (2014) proposed a framework that guides the use of blockchain technology from the perspective of storing and retrieving data associated with the forensic process. Weilbach (2014) already noted the use of having an immutable store for such mission critical information.

Weilbach (2014) focused on defining the problem and then setting out to perform a comprehensive literature study to determine the feasibility and practicality of the proposed solution. The scope of work presented by Weilbach (2014) is arguably slightly flawed, as is apparent through the following problem statement: "The research problem therefore, is that current logging efforts in support of IDS and Digital Forensic Readiness (DFR) lack a concurrent, secure and standardised means to communicate and store mission critical evidence". It is clear from this statement that the work is framed as improving IDS and logging capabilities only. Although not a misguided endeavour, it does somewhat limit the application of the technology to a very specific use case. The concept introduced could be more valuable if applied more generically and broadly.

Through conducting an extensive literature study, Weilbach (2014) identified that the area of study is very niche and that at the time there were not many academic works on the topic. This scarcity of resources has since subsequently changed with the growing popularity of blockchain and the research efforts associated with it. The related work of this very study is testament to the amount of research that has been conducted in the relevant field.

Weilbach (2014) proposed a framework and defined a basic protocol for implementing such a framework using the necessary technology. The protocol draws on related work in the form of logging standards and practices to define its own requirements. Although rudimentary, the protocol does advance the aim of having the solution standardised and accepted more widely.

2.3.4 Securing video integrity using decentralized trusted timestamping on the blockchain - Beel, Breitinger, Langer, Lommatzsch, and Gipp 2016

Beel *et al.* (2016) takes the concept of digital evidence and blockchains a step further by outlining and implementing an application for mobile phones that allows the user to store the signature of a video on the Bitcoin blockchain.

In the following extract it is easy to recognise how the work is very relevant to the conflation of blockchain and digital evidence: "The ability to verify the integrity of video files is important for consumer and business applications alike. Especially if video files are to be used as evidence in court, the ability to prove that a file existed in a certain state at a specific time and was not altered since is crucial. This paper proposes the use of blockchain technology to secure and verify the integrity of video files". What is notable here is the reference to admissibility of evidence in a legal context - also highlighted by Weilbach (2014). This solution is clearly geared toward a completely different use case, but with the same purpose in mind - validating the integrity of digital evidence.

Beel *et al.* (2016) reinforces the importance of having evidence of which the authenticity can be proven without a doubt. They continued by proposing a narrowly scoped solution for securely timestamping the digital signature of a piece of video evidence in the Bitcoin blockchain. In their work, Beel *et al.* (2016) also noted the versatility of blockchain technology and how it can potentially be applied to any domain where: "...a trustless [sic], anonymous, and tamperproof means of recordkeeping is required".

Timestamping of evidence, its history, and current applications is discussed in some depth to provide the reader with the necessary background to recognise the application of blockchain technology in this scenario. Beel *et al.* (2016) also explained the use and usefulness of timestamping from a practical and legal perspective.

Beel *et al.* (2016) based their work on previous work in the form of a timestamping service called OriginStamp. This web-based service allows its users to embed hashes of arbitrary data into the Bitcoin blockchain. The feasibility of providing such a service, given the estimated high cost of embedding data in the Bitcoin blockchain as seen in Okupski (2015), is questionable, but Beel *et al.* (2016) explained that the service is only possible as it performs one transaction of 1 Satoshi every 24 hours, embedding all hashes aggregated since the previous transaction.

Beel *et al.* (2016) proposed and developed a mobile application that has the ability to act as a Decentralised Trusted Timestamping (DTT) solution by leveraging the OriginStamp Application Programming Interface (API). Beel *et al.* (2016) considered that any blockchain can be used, and that their choice of the Bitcoin blockchain was merely the result of a lack of mature alternatives at the time of creating the DTT service. In the future work section of this paper there is a discussion about alternative applications of the solution proposed by Beel *et al.* (2016) that ranges from timestamping police body camera footage to CCTV and other aerial footage.

They also address the issue of admissibility in court by noting that, at the time of writing, no precedent has yet been established regarding the obligation of courts to recognise the evidence and its validity through DTT. Although the jurisdiction of the two instances certainly differ, the recent ruling that gives legal binding status to smart contracts and blockchain signatures in the American state of Arizona as noted by Campbell (2017), might indicate a shift in a positive direction for the acceptance of DTT practices around the world.

The alternative application of the technology, as noted in Beel *et al.* (2016), is recognised in this work, but it is proposed that the application of blockchain technology can be even further generalised to suit a wide array of use cases. The solution by Beel *et al.* (2016), is elegant and certainly uses the blockchain as intended in this work, but in a more limited scope.

2.3.5 Securing digital evidence information in Bitcoin - Wijaya and Suwarsono (2016)

Finally, the aptly titled: "Securing Digital Evidence Information in Bitcoin" by Wijaya and Suwarsono (2016) is examined as it directly addressed some of the concerns shared between the fields of digital forensics and blockchain technology.

Shortly after an introductory Bitcoin explanation, the discussion moves on to tax fraud and tax fraud investigations, where Wijaya and Suwarsono (2016) noted that digital evidence is often part of tax fraud investigations and that procedures exist for 'borrowing' this digital evidence from alleged perpetrators. They also noted that a letter describing this 'borrowing' of data was prepared and given to both the investigating authority and the subject of the investigation. On the contents of this document Wijaya and Suwarsono (2016) noted: "The digital data itself will be represented as hash values. The appendices

and the official letters are then signed by both parties: the tax investigators and the taxpayers. Both of the parties keep copies of the official letters.". From this it is clear that a possible use case for blockchain technology is developing.

Wijaya and Suwarsono (2016) continued by highlighting that the creation and distribution of this letter relies heavily on a trust-based system where there is currently ample opportunity for illegal modification of the data and that, as yet, no procedure exists for resolving a dispute or discrepancy between the copies of these letters and the data.

Wijaya and Suwarsono (2016) proposed a system where evidence hashes are incorporated into Bitcoin transactions and then embedded into the blockchain as a method of preserving and timestamping the evidence. This method would give an observer the ability to verify that the evidence they possess matches a hash of the original evidence. Wijaya and Suwarsono (2016) notes that in this use case, the purpose of the embedded hash is simply to prove that some arbitrary data existed at some point in time.

Wijaya and Suwarsono (2016) then offered a new perspective on the previously mentioned use case by proposing that the two parties both sign the data in question in the pursuit of non-repudiation. This method, unlike others observed before, relies on being able to link an identity to the embedded hash; for that to be possible both parties need to intrinsically link their identities to the hash by signing it with their public/private key pair.

The topic of economic feasibility is also addressed by Wijaya and Suwarsono (2016), when they calculated the cost for a single such transaction to be 10 000 Satoshi. Since two separate transactions need to be made, the cost of this scheme would be 20 000 Satoshi or (approximately USD 0.22 at the time of writing).

This work yet again highlighted how blockchain technology can be applied to solve problems of trust and integrity across a range of different scenarios.

2.3.6 Blockchain timestamping

A particularly relevant alternative application has been maturing over the last few years since Araoz and Ordano (2013) created the Proof of Existence (PoE) service. This service is said to have been the pioneer of what is now referred to as blockchain timestamping services (Wayne *et al.*, 2016).

The PoE service allowed a user to prove that some data existed at a certain point in time by embedding a hash of that data into the Bitcoin blockchain. As the creator of the

service noted on their website (Araoz and Ordano, 2013), the service aims to solve three problems:

- Demonstrating data ownership without revealing actual data
- Document timestamping
- Document integrity validation

Todd (2016b) explained four relevant use cases for the OpenTimestamps service (OTS), and discussed some of the benefits and challenges associated with these use cases.

The first use case Todd (2016b) explains, is very closely aligned with the purpose of this work and relates to record integrity. Specifically, Todd (2016b) notes that having an immutable source of timestamped logging data in the aftermath of a malicious network intrusion can streamline the investigation significantly. By having these timestamped records, logs, or backups, it can easily be confirmed if the data has been altered and enables the ability to narrow the focus of the investigation to a reduced time window. Todd (2016b) does provide a caveat to that statement by saying that this specific use case for timestamping data would not be beneficial if it is not known when such an intrusion occurred.

A second use case explained by Todd (2016b) is software signing and PGP, where they noted that timestamped software signatures embedded in the Bitcoin blockchain can serve as a historical record to validate, possibly expired, software signing proofs against.

The third use case is evidence authenticity, which again, aligns very narrowly with the purpose of this work. Todd (2016b) explained this use case by way of an example involving a website. If a website's content were to be hashed and timestamped at a point in time, it could be proven with certainty that some content existed on that website at that point in time. This could then be compared to a hash of archived content to validate the exact content on the website at that time. This has application in the legal realm as it could be used in lawsuits involving copyright and content distribution.

The fourth and final use case involved ownership. Todd (2016b) specifically noted that the initial motivation for the OTS project was to prove ownership and provenance. By keeping a record of deeds, titles, or sales receipts for high-value goods on the immutable blockchain, ownership and provenance can be verified and tracked over time. Again, Todd (2016b) highlighted a potential shortcoming of this use case when they noted that

a timestamped record does not prove the validity of items like a sales receipts; it merely limits the scope of potential fraud by giving investigators a narrower timeline to investigate within.

Another use case, as noted by Beel *et al.* (2016), is the DTT service mentioned in Section 2.3.4. Beel *et al.* (2016) proposed a scenario where video evidence of a traffic accident or incident is submitted to the DTT service in order to verify its existence and integrity in some future process concerning insurance or legal proceedings. Importantly, Beel *et al.* (2016) noted that: "Currently, there is no simple, cost-effective and automated method available to consumers to prove that video footage was not tampered with after a specific point in time. If the authenticity of a video file is contested, the status quo requires testimony of witnesses, or the hiring of experts to verify that the digital file has retained its integrity". This was reinforced shortly thereafter when referencing an incident involving forged satellite imagery and the downing of Malaysia Airlines Flight 17 (MH17) over the Ukraine. In this incident, it took almost two years to determine that the footage in question was tampered with. If DTTs were commonly used and accepted, this detection of tampering could have been near-instant.

Proof of Existence

Ultimately, PoE, or blockchain timestamping services makes use of the *scriptPubkey* method to embed the hash of arbitrary data into a provably un-spendable transaction output. This method is discussed in more detail when covering the OTS service below. By using this method, it is possible to embed a small amount of data permanently into the Bitcoin blockchain; the embedded data is prepended with some marker bytes that makes searching for such proofs in the blockchain easier. The marker bytes prepended to the hash are 0x444f4350524f4f46 which translates to 'DOCPROOF' using the American Standard Code for Information Interchange (ASCII) encoding.

The service is hosted as a web-based API that can be posted to from a browser or using a Command Line Interface (CLI) tool. The service was a major step forward in the development of blockchain-based timestamping and notarization services.

On the basis of the PoE model, similar services were developed, and today there are multiple services with a variety of different offerings and interaction models. Although the overall principle remains the same, these services vary in terms of size, cost, and performance; all of which have since been greatly improved upon. Additionally, as a

Table 2.4: Blockchain timestamping services

	Service Name					
	Proof of Existence	Open-Timestamps	Stampery	Chain-point	Eternity Wall	Origin Stamp
Uses the Bitcoin blockchain?	Y	Y	Y	Y	Y	Y
Uses the Ethereum blockchain?	N	N	Y	Y	N	N
Open Source (O), Proprietary (P) or Commercial (C)	P	O	P/C	P	P	P
Is a protocol?	N	Y	Y	Y	N	N
Is a service?	Y	Y	Y	Y	Y	Y

result of the popularisation of such services, a protocol for interacting with blockchain-based timestamping services has been developed. In the following section, these services and other advances will be discussed in more detail. Table 2.4 gives a quick overview of such services and some distinguishing properties of each.

From this comparison table, it can be seen that only OTS is a service and protocol and is open source, making it an exceedingly suitable candidate for further analysis and a possible candidate for use in this work.

OpenTimestamps

OTS developed by Todd (2016b) is an attempt to create a standardised protocol for timestamping information on the Bitcoin blockchain. The project is currently in the early phases of alpha development and is being actively maintained.

The OTS service consists of server-side and client-side components that interact to perform the timestamping of data as well as validate existing timestamps for which receipts 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 components then take the data and incorporate 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.

Todd (2016b) noted that the service has three distinct advantages over other timestamping services:

- Trust: By using the public Bitcoin blockchain it eliminates the need to use third parties or authorities to notarise data.
- Cost: OpenTimestamps scales by being able to create timestamps for vast amounts of data using a single low-value Bitcoin transaction.
- Convenience: OpenTimestamps has the ability to create a third-party verifiable timestamp in seconds.

Based on the above characteristics, the OTS offering appears very appealing in the current research domain.

Todd (2016b) explains that 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 by using the *nTime* block header field. The accuracy of such a time stamp is estimated by Todd (2016b) to be within two to three hours of the submission date and time. Since the *nTime* field is tightly coupled with the other block header fields containing the hash of the document, there is an inherent link between the data and the time, allowing any observer to verify that some arbitrary data existed at a specific time in the past.

Todd (2016b) noted that OTS also uses, what they term 'commitment operations'. A commitment operation can be any function that alters the function input to produce a deterministic output. A simple concatenation function such as $a \parallel b = ab$ 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. Some terminology - root, node and leaves - were discussed previously in Section 2.2.3. The usage of these terms is not coincidence

but rather as a result of the heavy reliance on MHT to support the OTS functionality, as discussed below.

Todd (2016b), like many others, recognised the issue of scalability (in terms of constrained data storage and speed) associated with the Bitcoin blockchain, and like others (Wijaya and Suwarsono, 2016), made use of various techniques to address these constraints. OTS primarily makes use of MHTs to address the problem of scalability but also employs other novel techniques like aggregation- and calendar services.

OTS embeds data on the Bitcoin blockchain by associating it with a Bitcoin transaction; more specifically, by embedding the hash of some known data into the output script field (*scriptPubkey* as noted in Table 2.3) of a transaction as a Bitcoin address. Since a transaction output has a limited amount of space available it would be impractical to store the large amounts of data in this field. Even if that data were hashed, having to create a transaction for many data sets would become expensive as a result of the fees associated with the many Bitcoin transactions necessary to accommodate hashes for each data set. Not only would the cost of these transaction be prohibitive, but large numbers of low value transactions could have a detrimental effect on the entire Bitcoin network by clogging it and slowing it down. 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 then collect all of the different hashes from different OTS clients, uses them as leaves of a MHT and computes the MHT. This root is in turn embedded into a single Bitcoin transaction.

Once a MHT root 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 $O(log_2(n))$ as noted in Table 2.1. Figure 2.6 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 the *scriptPubkey* Bitcoin transaction header by aggregating these leaves into a MHT. The root of that tree is then stored in a Bitcoin transaction, and returns only the commitments necessary to follow the commitment path up the tree and to the root. In Figure 2.6, only the blue nodes would need

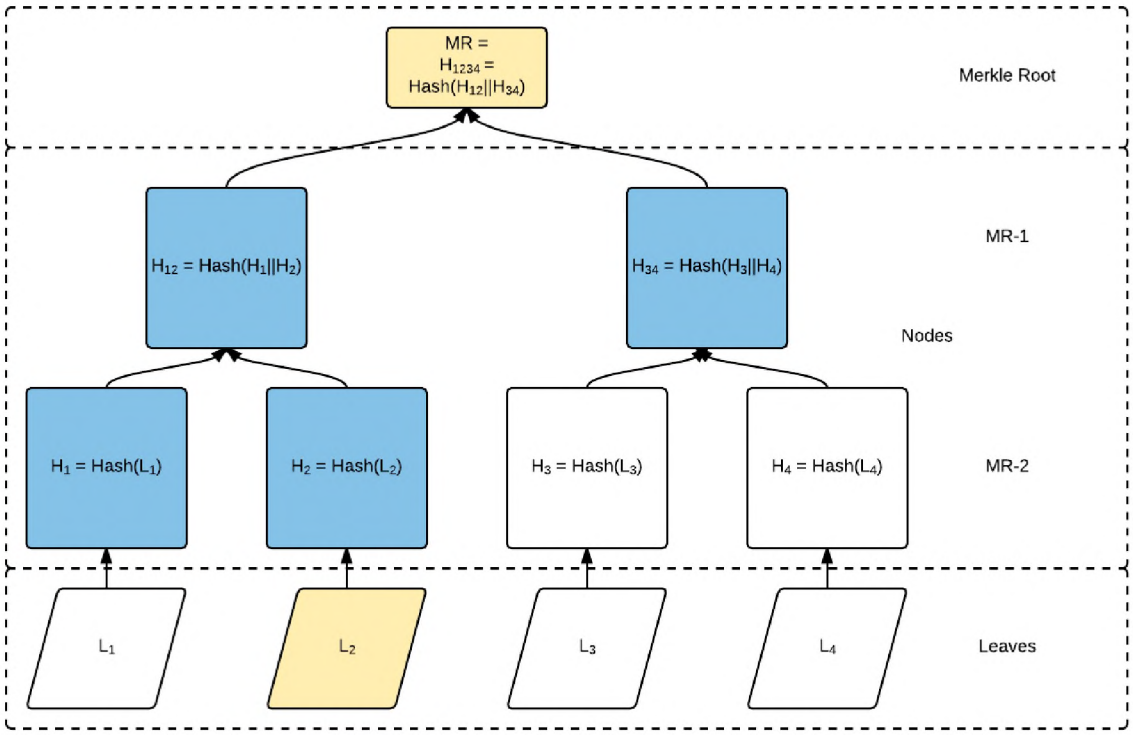


Figure 2.6: A series of relevant OTS commitment operations to verify leaf L_2

to be known by a client to enable it to replay the sequence of commitment operations to verify L_2. This does, however, rely on the assumption that the root (stored on the Bitcoin blockchain) and the leaf (the relevant original data), represented as yellow nodes, are already known by the client.

The MHT represented in Figure 2.6 can be considered a minimal example and MHTs can be much larger and complex, resulting in an increased storage efficiency as the efficiency is logarithmic ($O(log_2(n))$).

OTS further makes use of calendar servers to address the issue of speed, since aggregation and embedding into the Bitcoin blockchain may take too long for time sensitive notarization processes. Calendar servers act as an intermediary, confirming receipt of an attestation, and committing to having that attestation embedded into the Bitcoin blockchain at some point. Since calendar servers are completely under the control of the entity operating it, it does not afford the same assurances as the final proof but rather serves as a trade-off between convenience and security. They add the convenience of having an immediate proof but lack the immutability and security provided by the Bitcoin blockchain which takes time. A malicious calendar server cannot steal data since the data

sent to it is merely a hash of the original data, but one can falsely claim to commit the attestation to the Bitcoin blockchain on behalf of the aggregation server and then never do so. This would mean that the attestation would be lost and would have to be recommitted before a proof can be generated and verified. The risk is minimal and by having multiple calendar servers operational this risk can be mitigated by submitting redundant attestations.

2.4 Summary

Considering the above previous research on the topic of blockchains and digital evidence, there is a clear indication that the concept is becoming accepted and that many have identified how the key properties of blockchain technology can also be used to reinforce the integrity and validation of digital evidence.

All of the related work discussed above takes advantage of these key blockchain properties by using them to solve their own problem scenarios. No comment is made on the validity of one scenario over the other, but what is clear is that in all the examples mentioned there is a narrowly defined scope of application, be it video, political messages or tax documents. Conversely, there are very generic services like OriginStamp and OTS that allows the user thereof to embed the hash of any document in the Bitcoin blockchain. OTS, due to its open nature, is an ideal candidate to base further research upon. By utilizing an open standard like OTS, the adoption and acceptance of blockchain timestamping services in the legal context can be accelerated.

It is clear from the current state of research that there is ample opportunity to build on current implementations to develop a more generic and formalised approach to creating and validating digital evidence against an immutable public ledger. Furthermore, formalising such a system may lead to further development and accelerated acceptance of such systems in various legal jurisdictions.

Chapter 3

Research design

3.1 Research question

Given the current state of research into timestamping using decentralised trust systems and the overlap with the practice of hashing and verifying evidence as a proof of integrity, it seems prudent to explore how the application of blockchain technology can aid or better these practices. Following the literature review, it is also apparent that the best candidate technology for this would have to be transparent and open to encourage vetting and adoption. OTS as a protocol and an implementation of timestamping technology will therefore be best suited to achieve these research goals.

Apart from OTS as the candidate technology for notarisation and timestamping, the digital forensics software in which it will be implemented should also be accessible and open to support the integration of these technologies. Given these requirements, SleuthKit Autopsy emerges as an ideal candidate for merging these technologies because of its open source nature and extensibility discussed in Section 3.3.

The goal of this work can therefore be further crystallised by exploring the possibilities of implementing a new technology, OTS, to in widely used digital forensic tools, Autopsy, and measuring its effectiveness in aiding and maturing existing practices of evidence integrity verification.

3.2 Understanding OpenTimestamps

OTS, as alluded to in the literature review section of this work, is a new and novel implementation of blockchain technology to facilitate automated notarisation services without the reliance on a trusted central authority or third party. OTS aims to make these notarisation services accessible and transparent by using the Bitcoin blockchain to store and validate proofs.

It is essential that OTS be reviewed in detail to provide assurances as to its reliability and accuracy. This work seeks to promote the use of OTS for digital evidence integrity validation, and must assume that the integrity of the technology will at some point be called into question as part of an investigation or court proceedings. Therefore, it is necessary to provide a conclusive and vetted explanation of the proof mechanism to pass peer review. The aim of the following section is to provide a detailed review of OTS to support any potential future investigation into its suitability as a proof of existence and proof of integrity mechanism.

3.2.1 OTS timestamps

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 replayed, 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 the literature review section of this paper.

An OTS proof 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, namely:

- That the file existed in a specific time window in the past;
- That the file's content remains unmodified from the time the timestamp was created.

OTS utilises the immutability of the Bitcoin blockchain to remove the need for two or more parties to establish a mutual trust relationship with a third party to verify that any given attestation and its proof is genuine. It delegates the trust mechanism to the Bitcoin blockchain, which is inherently public along with all the operations, to support the attestations made regarding the state of the file. By doing this, the timestamp becomes immutable and independently verifiable to any concerned party looking to verify the existence integrity and timestamp of a specific file.

The exact operation to create a timestamp will be discussed in more detail in the coming sections, but for now it is useful to note that creating a timestamp for a file is called 'stamping' the file. The resultant timestamp, assuming the stamping process was successful, will be the original name of the stamped file with the *.ots* extension appended to indicate the OTS timestamp file type.

Below is an example of a complete OTS timestamp after being parsed and presented by an OTS utility:

```
File sha256 hash: bd7299df8b4c2717650fcfc9f409beffc454e9b7f201eec89f2de4fc0b535882
Timestamp:
append 0306d4367f450e71cb225b2e922aef94
sha256
 -> append 1c277205d32170fa9ac33ef24a562450
    sha256
    prepend 59aee87b
    append f4f65e1d23c9f037
    verify PendingAttestation('https://alice.btc.calendar.opentimestamps.org')
 -> append 372bfd2312ba2fb5109987241a229405
    sha256
    prepend 6e0bb638b20b762f51f4b63676a7f60665e9b3b85fa4122e950c8d18820871a3
    sha256
    prepend 59aee87a
    append cc8d9dc107815d8f
    verify PendingAttestation('https://finney.calendar.eternitywall.com')
 -> append f8f182995747b9c9ebce4cb40389cbd4
    sha256
    prepend 59aee87b
    append 6305c4687d0a20c4
    verify PendingAttestation('https://bob.btc.calendar.opentimestamps.org')
```

```
22    sha256
23    prepend 6fdb93f0a5e327a3acd274393961eddff9296cb0866673bf8b4dd4dad673c019
24    sha256
25    prepend 66ff25a8b732bc7b96623b8ac87cab579bc2869aee5cb9d5c0f6dc2ecee41b80
26    sha256
27    append 1f3dde8a14910392f613aaf271d46a840a21999c09c89ed1b962daf68a3578e7
28    sha256
29    append 269d331418a408ced27e3af285f4e44fb08c43283708286eb3f4932f4127ebf5
30    sha256
31    append 263b23d8aa562c2836ba5f4dbc641ac62e4018fb26c1fc31dbce0b595e8c8e0d
32    sha256
33    prepend 14ef2469176146f044e105f86385762d264443ebb98eecc56cb5984457de3972
34    sha256
35    append 3f59c91703dfc511d4977e8729d2a62e86d97303c3d0d10392f3a98cb13b6ec0
36    sha256
37    append de50842022d66983cea6637d78fea5032fe85f154632afa6a3ccd560551a5508
38    sha256
39    append 68b09f62ca7e0c5f7bf430830652dbba03078040e4926b7a6a0c2c0847c87eef
40    sha256
41    prepend 010000000118c8a478cddc58325969e1a409e7f2e4badfe57c12d25de2c8b73
42    b139e6050d10000000000fdffffff025abaf00000000000001600140db84d3cb80e3fe685
43    834583d6216d0736bc12660000000000000000226a20
44    append 6e610700
45    # Bitcoin transaction id 853b24b4cb03015c0781543c03710b4ecdb7db2319e511e44a6d279777
          ↪ f54895d
46    sha256
47    sha256
48    prepend d08ac122340781dc3507d97df99f2240044ea95f6a8701568d68b34c5167cb18
49    sha256
50    sha256
51    append 74e469a92c2662afa4ba63f6287806fd6af5db5f045ea5260abd4186799bc69e
52    sha256
53    sha256
54    append 66a91ddd3f81448f6c7ffd12be514fb558aa1e4b36bfe84b459111c45eff58bd
55    sha256
56    sha256
57    append ef757837405eb880bf3714316464e3520eac9503a681313d5084ad5c9bb93fd9
58    sha256
59    sha256
60    append fb9e30ce972c56810ebb62cbadb7cb593864354d7f1559665dc9baf7138e1d4a
61    sha256
62    sha256
63    append 12e216e71aa8ac191f3d4194d4010942a16f5d53377c9e8dd01e7420724b00c8
64    sha256
```

```
65      sha256
66      append 8e29aa8c4173bd7f1f0fd71c5305453ff81d7adb9fa709f0edda88f2f8ca375b
67      sha256
68      sha256
69      prepend 7ded30660ea096d34f8f10d1188124c84c01c81c198da47b2b74c305f96b9184
70      sha256
71      sha256
72      append 7a6d3c6ac2f4bd1077f5fe3f26f941cef92e1f8ab3776c196019adfeddda159d
73      sha256
74      sha256
75      prepend 2d7a16cd4a6b108ed6558fd28b195444f77c12d5d4b63275b53fb16c927ac87c
76      sha256
77      sha256
78      verify BitcoinBlockHeaderAttestation(483695)
79      # Bitcoin block merkle root 0f92e50cd5b32fa5c7c851b160daafca524aa9548a1ea7205249f
            ↪ c98d5b2014f
```

Todd (2016b) noted that a timestamp is essentially just a collection of commitment operations that are applied to an input message in a specified sequence, and that replaying those commitment operations in order is all that is necessary to verify the timestamp.

The basic anatomy of a timestamp can be divided into three main sections:

1. File hash
2. Merkle tree construction
3. Bitcoin block header attestation

Each of these vary significantly in size and complexity, but are equally important to the final timestamp verification.

What is seen above is an interpreted textual version of the timestamp commitment operations, but timestamps are binary data blobs which are not readily legible. Timestamps are saved in raw binary format to prevent issues with interpretation, encoding, and compatibility between systems. In email correspondence with the OTS author (can be seen in Section B.1), Todd (2017) noted that in a previous version of OTS, timestamps were not binary blobs, but instead looked similar what is shown below (using a Java Script Object Notation (JSON) structure):

```
"ops": [
      {
          "Hash": {
              "input": "13249541def3c688e28a32da9a6f39e2c68442db",
              "parents": [
                  [
                      0,
                      20
                  ]
              ],
              "algorithm": "sha256d",
              "digest": "49bdaf64146928c7ba30e5a28704e0762a37d53236438b4cd1d831f0568
                  ↪ b8535"
          }
      },
  ]
```

Todd (2017) eloborated by explaining that, contrary to what seems obvious, JSON is not the serialisation format of the timestamp above: the serialisation format of the timestamp is, in actual fact, a subset of the JSON standard, namely exactly what JSON elements are allowed in an OTS proof. For textual proofs, it is always necessary to parse the output of the textual parser itself, which introduces unnecessary complexity and leaves room for inaccurate interpretation. Since OTS is fundamentally security software, it is of critical importance that any interpreter completely understand the timestamp without misinterpretation or ambiguity.

By making the timestamp a raw binary format, OTS achieves two important goals that make the overall system more secure and trustworthy.

Firstly, it adds fragility to the timestamp interoperation to ensure that the smallest of misinterpretations would result in a completely invalid timestamp, and would be obvious to the interpreting system. This is achieved by having a strong 1-to-1 coupling between every single bit in the serialised timestamp and a component of the mathematical structure of the timestamp. This results in a system with very little redundancy, which is ideal for consensus-critical systems where even seemingly insignificant changes between two timestamps should result in different results. Todd (2017) noted: "For security software, this brittleness is a good thing, as we want incorrect implementations to fail completely, 100% of the time, rather than potentially give inaccurate results."

Secondly, by storing the timestamp in a raw binary format, OTS reduces the potential

size of the timestamp since space can be saved by not having a universal and versatile schema definition and markup associated with formats such as JSON and eXtensible Markup Language (XML). This also helps with fragility, in that these schema definitions, of formats like JSON and XML, can be more forgiving and will tolerate some errors depending on the implementation of the interpreter.

A timestamp is clearly a complex artefact as can be seen above. Of course, a timestamp on its own is meaningless and needs the be verified to provide any value. As with the stamping operation, the verification operation, along with many more OTS operations, will be discussed in much more detail in the coming sections, but below is an example of the verification operation output for the above stated timestamp:

```
1 Success! Bitcoin attests data existed as of Tue Sep 5 20:24:51 2017 CEST
```

As is clear from the verification output, OTS was able to attest that the source file in question existed in its current state as of Tuesday, September 5 20:24:51 2017 CEST. It is important to note that the verification is clear and succinct, and leaves little room for interpretation and ambiguity, which is ideal when considering the intended use case of proving integrity of digital evidence.

The date in the timestamp verification confirmation is accurate up to the second, and the accuracy of this statement will be the subject of much more detailed analysis and discussion.

3.2.2 OTS implementations and dependencies

Before the subtleties and intricacies of OTS is discussed, note that OTS was designed to be widely adopted and compatible, and has therefore been implemented in a range of languages and frameworks to ensure its continued adoption and development.

As can be seen from the Github repository at Opentimestamps (2017), the OTS protocol has a wide range of different implementations. These are:

- opentimestamps-client: The OTS client component to create and verify OTS proofs in Python
- python-opentimestamps: The core OTS libraries used by both server and client components in Python

- javascript-opentimestamps: The implementation of OTS based on the Python implementation at Opentimestamps-Python and Opentimestamps-Client in JavaScript
- opentimestamps.org: The OTS (OpenTimestamps.org) website
- java-opentimestamps: The implementation of OTS based on the python implementation at
  Opentimestamps-Python and Opentimestamps-Client in Java
- opentimestamps-server: OpenTimestamps server component in Python
- rust-opentimestamps: The library for OTS in Rust

The focus of this section will, for the sake of simplification, centre on the most recent Python implementations of OTS at the time of writing (opentimestamps-client-v0.5.0, python-opentimestamps v0.1.0 and opentimestamps-server-v0.1.2).

3.2.3 OTS functions

Before exploring the lifecycle of an OTS proof, it is prudent to first obtain a better understanding of the functionality extended to the user of the opentimestamps-client. By looking at the functions and seeing how to invoke them and what output they produce, the reader will get a much clearer idea of how the protocol is intended to be used.

A discussion about the setup procedure and configuration of the OTS client-side components are outside the scope of this work and is very well documented in the code repository of the latest versions of the respective components. The setup process is automated and presents a low barrier to entry for less tech-savvy users. The functions shown will be invoked via the OTS CLI interface, but it is worth noting that these same commands can be wrapped by a Graphical User Interface (GUI) utility for users who prefer such a tool. The underlying functions, however, remain the same.

To illustrate the various OTS functions, a simple test text file was created. This file is called *testots.txt*. The content of the file will be a short sentence: "This is a test file."

```
1 user@host:~/otsdemo$ cat testots.txt
2 This is a test file.
```

By invoking the OTS client without supplying any parameters, the following usage guide - which indicates that the Stamp function can be invoked with the *s* argument, followed by the file to be stamped - is produced:

```
user@host:~/otsdemo$ ots
usage: ots [-h] [--version] [-q] [-v] [-l URL] [--no-default-whitelist]
           [--cache CACHE_PATH | --no-cache]
           [--btc-testnet | --btc-regtest | --no-bitcoin] [-w]
           [--socks5-proxy SOCKS5_PROXY] [--bitcoin-node BITCOIN_NODE]
           {stamp,s,upgrade,u,verify,v,info,i,git-extract} ...
```

There is also a more detailed version of the help function that can be invoked by calling OTS with the *-h* flag, which produces the following output:

Listing 3.1: Verbose output of the OTS help function

```
user@host:~/otsdemo$ ots -h
usage: ots [-h] [--version] [-q] [-v] [-l URL] [--no-default-whitelist]
           [--cache CACHE_PATH | --no-cache]
           [--btc-testnet | --btc-regtest | --no-bitcoin] [-w]
           [--socks5-proxy SOCKS5_PROXY] [--bitcoin-node BITCOIN_NODE]
           {stamp,s,upgrade,u,verify,v,info,i,git-extract} ...

OpenTimestamps client.

optional arguments:
  -h, --help show this help message and exit
  --version show program''s version number and exit
  -q, --quiet Be more quiet.
  -v, --verbose Be more verbose. Both -v and -q may be used multiple
                      times.
  -l URL, --whitelist URL
                      Add a calendar to the whitelist.
  --no-default-whitelist
                      Do not load the default remote calendar whitelist;
                      contact only calendars that have been manually added
                      with --whitelist
  --cache CACHE_PATH Location of the timestamp cache. Default: ~/.cache/ots
  --no-cache Disable the timestamp cache
  --btc-testnet Use Bitcoin testnet rather than mainnet
  --btc-regtest Use Bitcoin regtest rather than mainnet
  --no-bitcoin Disable Bitcoin entirely
  -w, --wait When creating, upgrading, or verifying timestamps,
                      wait until a complete timestamp committed in the
                      Bitcoin blockchain is available instead of returning
```

```
                      immediately.
  --socks5-proxy SOCKS5_PROXY
                      Route all traffic through a socks5 proxy, including
                      DNS queries. The default port is 1080. Format:
                      domain[:port] (e.g. localhost:9050)
  --bitcoin-node BITCOIN_NODE
                      Bitcoin node URL to connect to (defaults to local
                      configuration)

Subcommands:
  All operations are done through subcommands:

  {stamp,s,upgrade,u,verify,v,info,i,git-extract}
    stamp (s) Timestamp files
    upgrade (u) Upgrade remote calendar timestamps to be locally
                      verifiable
    verify (v) Verify a timestamp
    info (i) Show information on a timestamp
    git-extract Extract timestamp for a single file from a timestamp
                      git commit
```

For the sake of detail when demonstrating the different function calls, all calls are made with the *-v* flag set, which enables verbose output.

The **Stamp** operation, which is logically the first operation a user of OTS would perform, invokes the Stamp function which produces the timestamp that can later be verified. Calling the Stamp function is depicted below and can be done by invoking OTS with the *s* subcommand:

```
user@host:~/otsdemo$ ots -v s testots.txt
Doing 2-of-3 request, timeout is 5 seconds
Submitting to remote calendar https://a.pool.opentimestamps.org
Submitting to remote calendar https://b.pool.opentimestamps.org
Submitting to remote calendar https://a.pool.eternitywall.com
1.66 seconds elapsed
```

This function produces the initial timestamp and saves the *testots.txt.ots* file. It is important to note that this is an initial, or incomplete, timestamp and that there are further actions to be taken to make it a complete timestamp.

```
user@host:~/otsdemo$ ls -l
total 8
-rw-rw-r-- 1 user user 21 Oct 3 19:05 testots.txt
```

```
-rw-rw-r-- 1 user user 352 Oct 3 19:47 testots.txt.ots
```

The second function is **Info**, which parses the timestamp and displays information about it. The Info function can be executed by using the *i* subcommand and supplying the timestamp file as input argument:

```
user@host:~/otsdemo$ ots -v i testots.txt.ots
File sha256 hash: 649b8b471e7d7bc175eec758a7006ac693c434c8297c07db15286788c837154a
Timestamp:
append b148f67dd8c0081046b196cb5aa8dcc2 == 649b8b471e7d7bc175eec758a7006ac693c434c8297
    ↪ c07db15286788c837154ab148f67dd8c0081046b196cb5aa8dcc2
sha256 == bd2c8ac682b8ed4b544ddd29ce229ef42479162b1ff14cdd51c653601600b40b
 -> append 4bc7414f9b79e5b2a9699a15449f79d8 == bd2c8ac682b8ed4b544ddd29ce229ef42479162
     ↪ b1ff14cdd51c653601600b40b4bc7414f9b79e5b2a9699a15449f79d8
    sha256 == 04b5b76515735a80be9b465887d2f83d5423bf5e3540741e6e1ca5be78728e93
    prepend 59d3cd30 == 59d3cd3004b5b76515735a80be9b465887d2f83d5423bf5e3540741e6e1ca5b
        ↪ e78728e93
    append 68a95ad6aeade4bd == 59d3cd3004b5b76515735a80be9b465887d2f83d5423bf5e3540741
        ↪ e6e1ca5be78728e9368a95ad6aeade4bd
    verify PendingAttestation('https://finney.calendar.eternitywall.com')
 -> append 782372bb88a18335daf2a8e596338454 == bd2c8ac682b8ed4b544ddd29ce229ef42479162
     ↪ b1ff14cdd51c653601600b40b782372bb88a18335daf2a8e596338454
    sha256 == 15bc3053e59447106c4b233c36336389e7e6b5ee7a625121313c7a0b0ebbc75e
    prepend 59d3cd30 == 59d3cd3015bc3053e59447106c4b233c36336389e7e6b5ee7a625121313c7a0
        ↪ b0ebbc75e
    append 50d3d70950797d54 == 59d3cd3015bc3053e59447106c4b233c36336389e7e6b5ee7
        ↪ a625121313c7a0b0ebbc75e50d3d70950797d54
    verify PendingAttestation('https://alice.btc.calendar.opentimestamps.org')
 -> append e97b3ae7e7270e1778c5e596cc440842 == bd2c8ac682b8ed4b544ddd29ce229ef42479162
     ↪ b1ff14cdd51c653601600b40be97b3ae7e7270e1778c5e596cc440842
    sha256 == d2146113dcd4fbdc95b3f4cb984d59a78ead957d07bca92ddc8a0439dc4aa5ff
    prepend 59d3cd30 == 59d3cd30d2146113dcd4fbdc95b3f4cb984d59a78ead957d07bca92ddc8
        ↪ a0439dc4aa5ff
    append 75cc0807f01f2591 == 59d3cd30d2146113dcd4fbdc95b3f4cb984d59a78ead957d07bca92d
        ↪ dc8a0439dc4aa5ff75cc0807f01f2591
    verify PendingAttestation('https://bob.btc.calendar.opentimestamps.org')
```

Once the timestamp attestation has been generated, it takes some time for it to be incorporated into the Bitcoin blockchain by OTS (this process will be discussed in more detail in the following section). Once this happens, the timestamp needs to be upgraded to reflect this commitment to the Bitcoin blockchain to form the final timestamp. The **Upgrade** function can be run by invoking OTS with the *u* subcommand:

```
1  user@host:~/otsdemo$ ots -v u testots.txt.ots
2  Upgrading testots.txt.ots
3  Checking calendar https://alice.btc.calendar.opentimestamps.org for 59d3cd3015bc3053
       ↪ e59447106c4b233c36336389e7e6b5ee7a625121313c7a0b0ebbc75e50d3d70950797d54
4  Got 1 attestation(s) from https://alice.btc.calendar.opentimestamps.org
5      BitcoinBlockHeaderAttestation(488163)
6  Checking calendar https://finney.calendar.eternitywall.com for 59d3cd3004b5b76515735
       ↪ a80be9b465887d2f83d5423bf5e3540741e6e1ca5be78728e9368a95ad6aeade4bd
7  Got 1 attestation(s) from https://finney.calendar.eternitywall.com
8      BitcoinBlockHeaderAttestation(488163)
9  Checking calendar https://bob.btc.calendar.opentimestamps.org for 59d3cd30d2146113dcd4
       ↪ fbdc95b3f4cb984d59a78ead957d07bca92ddc8a0439dc4aa5ff75cc0807f01f2591
10 Got 1 attestation(s) from https://bob.btc.calendar.opentimestamps.org
11     BitcoinBlockHeaderAttestation(488163)
12 Got new timestamp data; renaming existing timestamp to 'testots.txt.ots.bak'
13 Success! Timestamp complete
```

As is clear from the output of the Upgrade function, the timestamp is now complete and ready to be verified. Similar to previous functions, **Verify** can be called by invoking OTS with a subcommand, in this case *v*:

```
1  user@host:~/otsdemo$ ots -v v testots.txt.ots
2  Assuming target filename is 'testots.txt'
3  Hashing file, algorithm sha256
4  Got digest 649b8b471e7d7bc175eec758a7006ac693c434c8297c07db15286788c837154a
5  Attestation block hash: 000000000000000000031944aee9496e6c77f909508b797b19b9f6a662a6
       ↪ e6996
6  Success! Bitcoin attests data existed as of Tue Oct 3 20:15:45 2017 CEST
```

The above examples cover the most basic of OTS functionality and the logical order in which functions can be executed to generate and verify an OTS timestamp. This is not to say that this sequence will always be followed or that the results will always be the same. OTS can, for instance, be configured to submit file hashes to custom aggregation servers, using proxies, using a local cache, using testnet (the Bitcoin test network) etc.

In the above examples, no custom configuration was done and OTS was executed with the default configuration in place.

3.2.4 OTS 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 uses a system 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 lifecycle. It is, therefore, important that an exploration 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 two trust domains.

We'll designate 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: not SELF nor BTC

Ideally, instances where OTHER is trusted needs 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. 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

Table 3.1: Relevant trust domains per configuration option

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

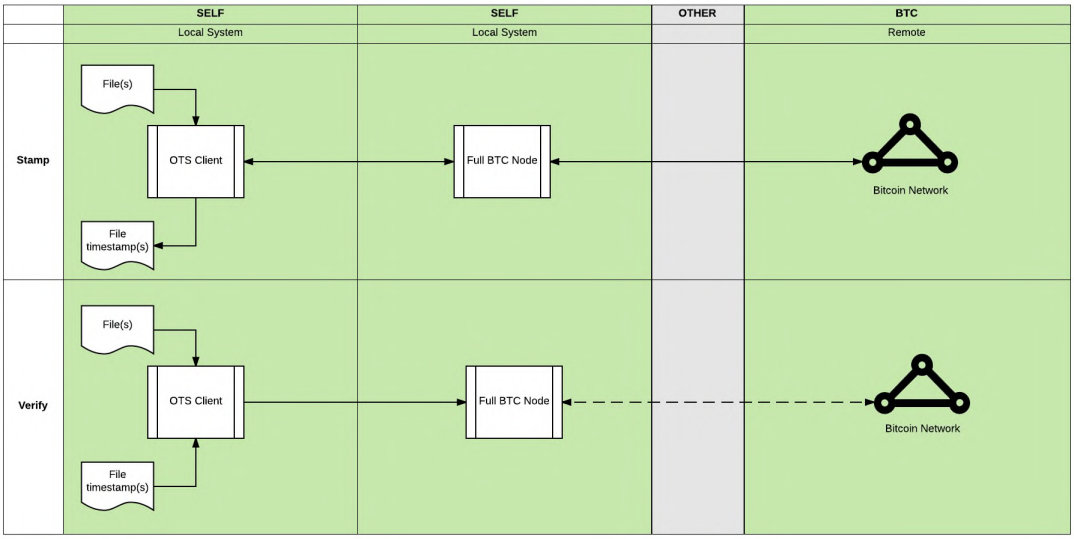


Figure 3.1: Major components of OTS in trust domains for Configuration A

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 3.1.

Configuration A, being fully trusted, is depicted in Figure 3.1. 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 Figure 3.1 would require increased effort to configure and run, as all the components would have to be installed by the user. Additionally, this configuration would also carry a cost to the user, since they would be responsible for the

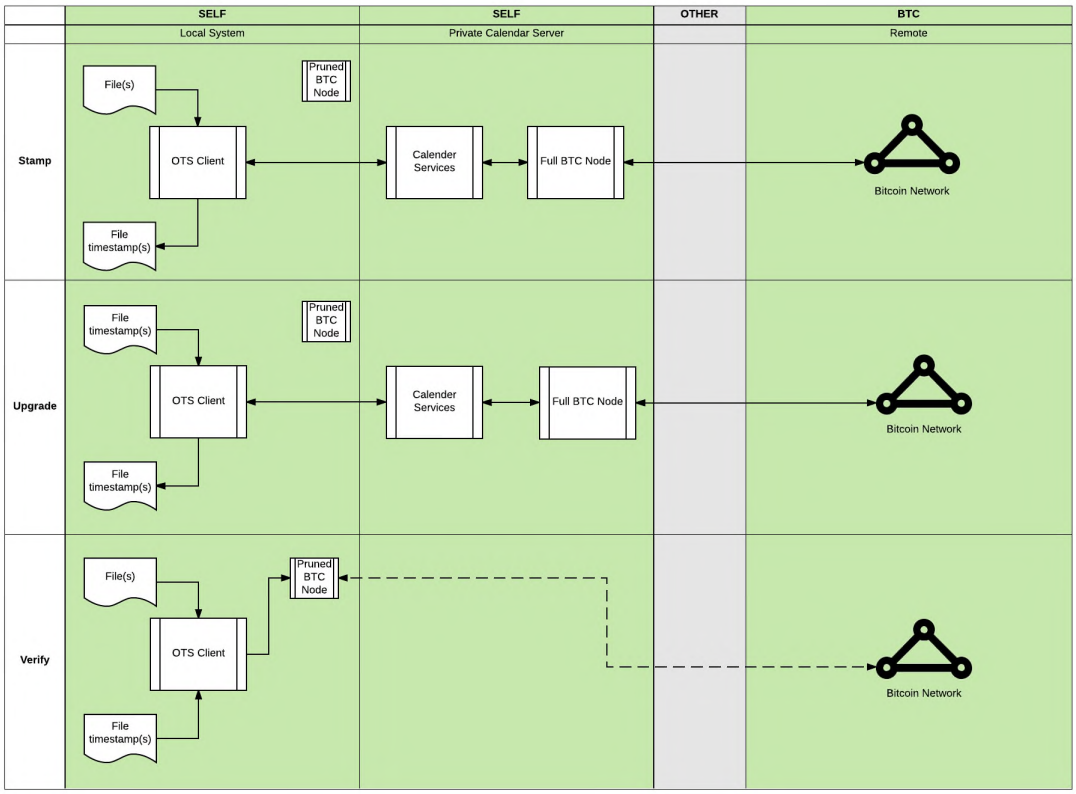


Figure 3.2: Major components of OTS in trust domains for Configuration B

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 Figure 3.2. 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 Figure 3.3 is semi-trusted as it includes the OTHER trust domain by making use of public calendar servers. The configuration of C is very

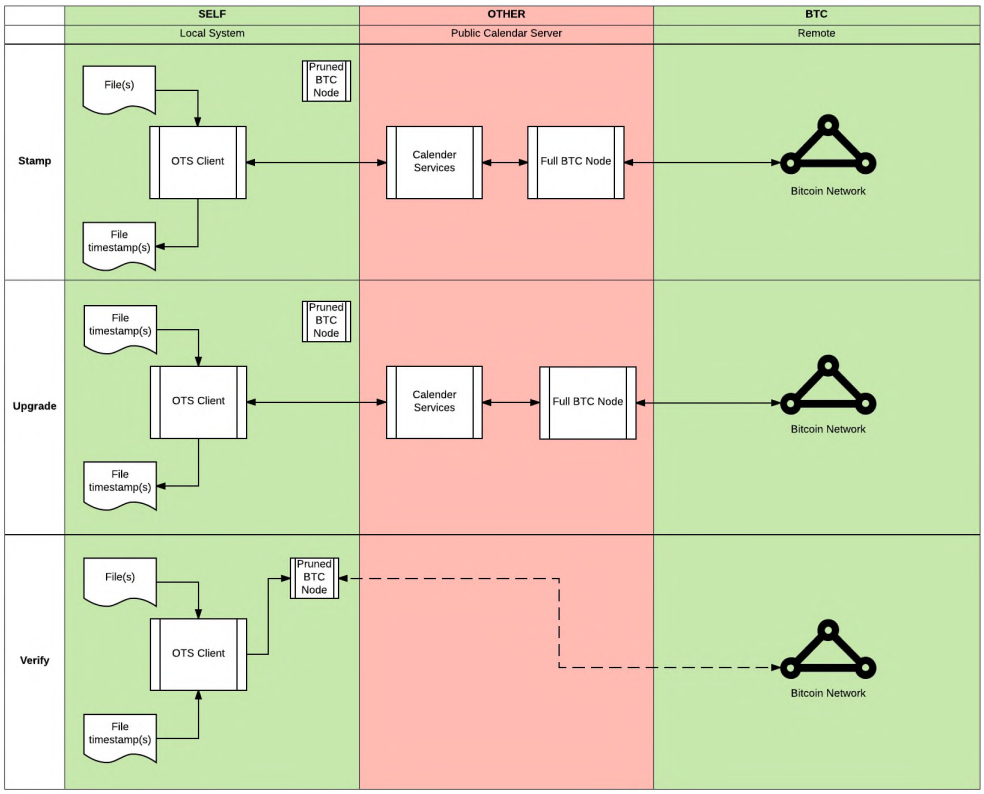


Figure 3.3: Major components of OTS in trust domains for Configuration C

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.

3.2.5 OTS timestamp lifecycle

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 Figure 3.3.

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 (Bitcoin Foundation, 2016).

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.

Using the same file (*testots.txt*) as in the previous example, a detailed description of the processes and systems involved in each of the core OTS functions is given below.

Stamp

When stamping a file, the OTS client generates a *SHA256* hash H of the target file.

A MHT is constructed with H to produce a *(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.

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 supplied 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*]

A populated version of this request will look as follows:

https://finney.calendar.eternitywall.com/digest/59d3cd3004b5b76515735a80be9b465887d2f83d5423bf5e3540741e6e1ca5be78728e9368a95ad6aeade4bd

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 as illustrated in Figure 3.4 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

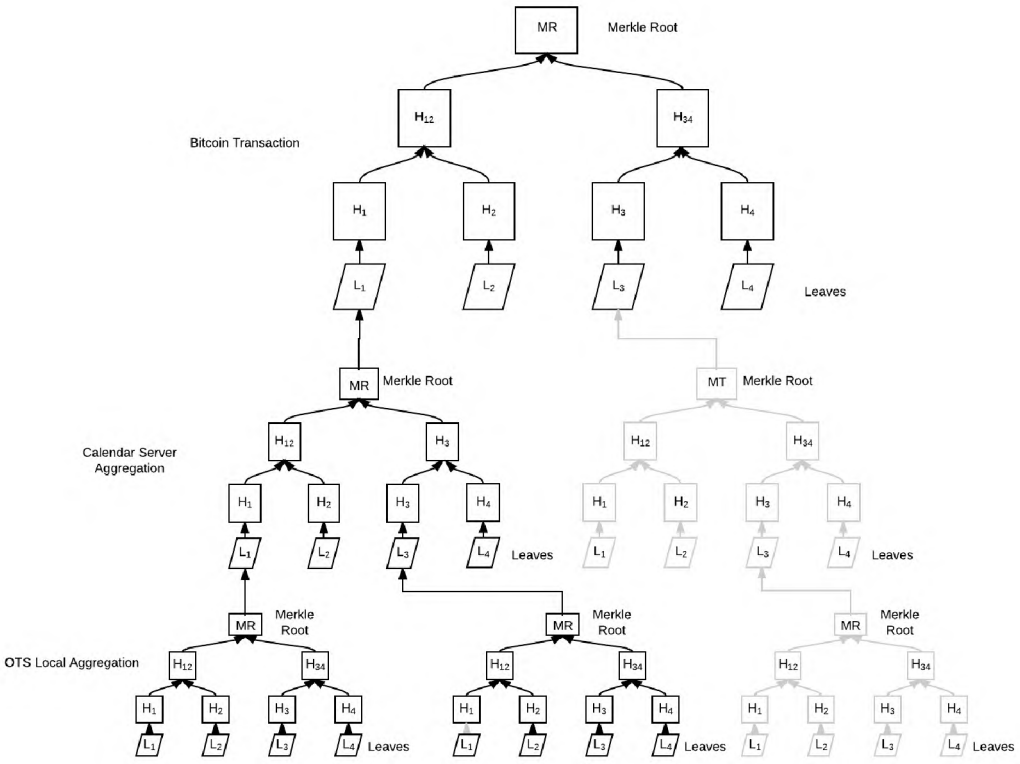


Figure 3.4: Nested MHTs on various aggregation levels

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 retuned timestamp will contain the relevant commitment operations and timestamp identifier for each calendar server that committed to submitting the timestamp. This commitment by the calendar server can be seen in the output below:

Listing 3.2: A parsed, incomplete timestamp example

```
1 user@host:~/otsdemo$ ots -v i testots.txt.ots
2 File sha256 hash: 649b8b471e7d7bc175eec758a7006ac693c434c8297c07db15286788c837154a
3 Timestamp:
4 append b148f67dd8c0081046b196cb5aa8dcc2 == 649b8b471e7d7bc175eec758a7006ac693c434c8297
      ↪ c07db15286788c837154ab148f67dd8c0081046b196cb5aa8dcc2
5 sha256 == bd2c8ac682b8ed4b544ddd29ce229ef42479162b1ff14cdd51c653601600b40b
6  -> append 4bc7414f9b79e5b2a9699a15449f79d8 == bd2c8ac682b8ed4b544ddd29ce229ef42479162
       ↪ b1ff14cdd51c653601600b40b4bc7414f9b79e5b2a9699a15449f79d8
7     sha256 == 04b5b76515735a80be9b465887d2f83d5423bf5e3540741e6e1ca5be78728e93
8     prepend 59d3cd30 == 59d3cd3004b5b76515735a80be9b465887d2f83d5423bf5e3540741e6e1ca5b
```

```
        ↪ e78728e93
9     append 68a95ad6aeade4bd == 59d3cd3004b5b76515735a80be9b465887d2f83d5423bf5e3540741
        ↪ e6e1ca5be78728e9368a95ad6aeade4bd
10    verify PendingAttestation('https://finney.calendar.eternitywall.com')
11  -> append 782372bb88a18335daf2a8e596338454 == bd2c8ac682b8ed4b544ddd29ce229ef42479162
      ↪ b1ff14cdd51c653601600b40b782372bb88a18335daf2a8e596338454
12    sha256 == 15bc3053e59447106c4b233c36336389e7e6b5ee7a625121313c7a0b0ebbc75e
13    prepend 59d3cd30 == 59d3cd3015bc3053e59447106c4b233c36336389e7e6b5ee7a625121313c7a0
        ↪ b0ebbc75e
14    append 50d3d70950797d54 == 59d3cd3015bc3053e59447106c4b233c36336389e7e6b5ee7
        ↪ a625121313c7a0b0ebbc75e50d3d70950797d54
15    verify PendingAttestation('https://alice.btc.calendar.opentimestamps.org')
16  -> append e97b3ae7e7270e1778c5e596cc440842 == bd2c8ac682b8ed4b544ddd29ce229ef42479162
      ↪ b1ff14cdd51c653601600b40be97b3ae7e7270e1778c5e596cc440842
17    sha256 == d2146113dcd4fbdc95b3f4cb984d59a78ead957d07bca92ddc8a0439dc4aa5ff
18    prepend 59d3cd30 == 59d3cd30d2146113dcd4fbdc95b3f4cb984d59a78ead957d07bca92ddc8
        ↪ a0439dc4aa5ff
19    append 75cc0807f01f2591 == 59d3cd30d2146113dcd4fbdc95b3f4cb984d59a78ead957d07bca92d
        ↪ dc8a0439dc4aa5ff75cc0807f01f2591
20    verify PendingAttestation('https://bob.btc.calendar.opentimestamps.org')
```

The timestamp identifier for `https://finney.calendar.eternitywall.com`, `https://alice.btc.calendar.opentimestamps.org` and `https://bob.btc.calendar.opentimestamps.org` will be '**59d3cd3004...de4bd**' on line 9, '**59d3cd3015...797d54**' on line 14 and '**59d3cd30d2...f2591**' on line 19 respectively.

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

Info

The simplest of all the OTS functions is the function which takes any timestamp as input, parses the commitment operations 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 operations 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; it also 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.

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]*

Which, when populated with values, will look as follows:

https://finney.calendar.eternitywall.com/timestamp/59d3cd3004b5b76515735a80be9b465887d2f83d5423bf5e3540741e6e1ca5be78728e9368a95ad6aeade4bd

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.

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.

Below is an example of a complete timestamp which is locally verifiable after being parsed in verbose mode via the Info function:

Listing 3.3: A parsed, complete timestamp example

```
1  user@host:~/otsdemo$ ots -v i testots.txt.ots
2  File sha256 hash: 649b8b471e7d7bc175eec758a7006ac693c434c8297c07db15286788c837154a
3  Timestamp:
4  append b148f67dd8c0081046b196cb5aa8dcc2 == 649b8b471e7d7bc175eec758a7006ac693c434c8297
       ↪ c07db15286788c837154ab148f67dd8c0081046b196cb5aa8dcc2
5  sha256 == bd2c8ac682b8ed4b544ddd29ce229ef42479162b1ff14cdd51c653601600b40b
6   -> append 4bc7414f9b79e5b2a9699a15449f79d8 == bd2c8ac682b8ed4b544ddd29ce229ef42479162
       ↪ b1ff14cdd51c653601600b40b4bc7414f9b79e5b2a9699a15449f79d8
7      sha256 == 04b5b76515735a80be9b465887d2f83d5423bf5e3540741e6e1ca5be78728e93
8      prepend 59d3cd30 == 59d3cd3004b5b76515735a80be9b465887d2f83d5423bf5e3540741e6e1ca5b
           ↪ e78728e93
9      append 68a95ad6aeade4bd == 59d3cd3004b5b76515735a80be9b465887d2f83d5423bf5e3540741
           ↪ e6e1ca5be78728e9368a95ad6aeade4bd
10     verify PendingAttestation('https://finney.calendar.eternitywall.com')
```

```
-> append 782372bb88a18335daf2a8e596338454 == bd2c8ac682b8ed4b544ddd29ce229ef42479162
    ↪ b1ff14cdd51c653601600b40b782372bb88a18335daf2a8e596338454
   sha256 == 15bc3053e59447106c4b233c36336389e7e6b5ee7a625121313c7a0b0ebbc75e
   prepend 59d3cd30 == 59d3cd3015bc3053e59447106c4b233c36336389e7e6b5ee7a625121313c7a0
       ↪ b0ebbc75e
   append 50d3d70950797d54 == 59d3cd3015bc3053e59447106c4b233c36336389e7e6b5ee7
       ↪ a625121313c7a0b0ebbc75e50d3d70950797d54
   verify PendingAttestation('https://alice.btc.calendar.opentimestamps.org')
   sha256 == 56367592cd684c5c2a03e71d353173a921f2441c9ff957cb1bdb926caed295a3
   append 716f6fe6011712e3556d65265b82eebd545de43b7da9f58530da1e9139251bac == 56367592
       ↪ cd684c5c2a03e71d353173a921f2441c9ff957cb1bdb926caed295a3716f6fe6011712e3556
       ↪ d65265b82eebd545de43b7da9f58530da1e9139251bac
   ...[29 lines elided]...
   8d7d1239063b8ba73cee03255f13c19f8ecb6911cb3c9c80de614cf363284ccb45e6f25d147c1d93045
       ↪ f437786e02157cf03d6f63aa8b1a201eeaa8194250fd1
   sha256 == 37fa6f0f61f4fc61da240549f87f5aff48b2e633e1be9020fbb2b8ecf747bd67
   prepend 010000000169845a160e4f3ec0959ac5911a84dea65519713c9ce52bf707197f3ae0ab02
       ↪ e70000000000fdffffff02a0385e0000000000160014e200dde45eb0529aebe86e16060fb9
       ↪ b109008b56000000000000000000226a20 == 010000000169845a160e4f3ec0959ac5911a84de
       ↪ a65519713c9ce52bf707197f3ae0ab02e70000000000fdffffff02a0385e0000000000160014
       ↪ e200dde45eb0529aebe86e16060fb9b109008b56000000000000000000226a2037fa6f0f61f4f
       ↪ c61da240549f87f5aff48b2e633e1be9020fbb2b8ecf747bd67
   append e2720700 == 010000000169845a160e4f3ec0959ac5911a84dea65519713c9ce52bf707197
       ↪ f3ae0ab02e70000000000fdffffff02a0385e0000000000160014e200dde45eb0529aebe86
       ↪ e16060fb9b109008b56000000000000000000226a2037fa6f0f61f4fc61da240549f87f5aff48
       ↪ b2e633e1be9020fbb2b8ecf747bd67e2720700
   # Bitcoin transaction id 9fd255b44d2373d98382a5469bda07862e1c1f6b89b5a4d7750309d958
       ↪ ce8809
   sha256 == 330ef1825b2394ea48b89e963296e375d139afdaa73f17773ef33f697f2fae09
   sha256 == 0988ce58d9090375d7a4b5896b1f1c2e8607da9b46a58283d973234db455d29f
   prepend 85028d0e6bc68dd101f5b3394db660777faa02ef1c7bfdd0440f8ce569e63b4f == 85028d0
       ↪ e6bc68dd101f5b3394db660777faa02ef1c7bfdd0440f8ce569e63b4f0988ce58d9090375d7
       ↪ a4b5896b1f1c2e8607da9b46a58283d973234db455d29f
   ...[39 lines elided]...
   d89cf9425ed1905ea9ce2fac1a3256e8cb26de834732070186ec38bf41e59cc237f4d3b525e4b4aaa
       ↪ f28798014e234f4336964dc728e54d6f40718c231439919
   sha256 == 57414fcb4efefd124f2777f1d07043a97407ca09371cfa98bfcf16834f10f43e
   sha256 == 34c7c3b00355d8f5fb7dfd9441c3cab033fc94c80442a0466f4143867c1e49a7
   verify BitcoinBlockHeaderAttestation(488163)
   # Bitcoin block merkle root a7491e7c8643416f46a04204c894fc33b0cac34194fd7dfbf5
       ↪ d85503b0c3c734
-> append e97b3ae7e7270e1778c5e596cc440842 == bd2c8ac682b8ed4b544ddd29ce229ef42479162
    ↪ b1ff14cdd51c653601600b40be97b3ae7e7270e1778c5e596cc440842
   sha256 == d2146113dcd4fbdc95b3f4cb984d59a78ead957d07bca92ddc8a0439dc4aa5ff
```

```
35    prepend 59d3cd30 == 59d3cd30d2146113dcd4fbdc95b3f4cb984d59a78ead957d07bca92ddc8
          ↪ a0439dc4aa5ff
36    append 75cc0807f01f2591 == 59d3cd30d2146113dcd4fbdc95b3f4cb984d59a78ead957d07bca92d
          ↪ dc8a0439dc4aa5ff75cc0807f01f2591
37    verify PendingAttestation('https://bob.btc.calendar.opentimestamps.org')
```

Note the significant difference in size and complexity between an incomplete timestamp in Listing 3.2, and the complete timestamp in Listing 3.3. This size 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.

Also note that there are still three distinct commitments starting on lines 6, 11 and 79 in Listing 3.3. This is because the initial timestamp was submitted to three calendar servers as a redundancy mechanism, and that the complete timestamp was retrieved only from `https://alice.btc.calendar.opentimestamps.org` starting at line 11 and ending at line 78. The complete timestamps were not retrieved from the other calendar servers as one valid timestamp is sufficient to perform local verification.

The command for verification looks as follows:

Listing 3.4: Verification of timestamp testots.txt.ots

```
1  user@host:~/otsdemo$ ots -v v testots.txt.ots
2  Assuming target filename is 'testots.txt'
3  Hashing file, algorithm sha256
4  Got digest 649b8b471e7d7bc175eec758a7006ac693c434c8297c07db15286788c837154a
5  Attestation block hash: 000000000000000000031944aee9496e6c77f909508b797b19b9f6a662a6
       ↪ e6996
6  Success! Bitcoin attests data existed as of Tue Oct 3 20:15:45 2017 CEST
```

Below is a step-by-step walkthrough of exactly how this timestamp was verified, and how it was possible to make the attestation that it did. 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.

Step 1 - Calculate file *sha256* signature:

Listing 3.5: Extract from Listing 3.3 line 2

```
1  File sha256 hash: 649b8b471e7d7bc175eec758a7006ac693c434c8297c07db15286788c837154a
```

Description: 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.

Manual reproduction:

Listing 3.6: Manual reproduction of hash operation

```
1 user@host:~/otsdemo$ openssl dgst -sha256 testots.txt
2 SHA256(testots.txt)= 649b8b471e7d7bc175eec758a7006ac693c434c8297c07db15286788c837154a
```

Step 2 - Local noncing:

Listing 3.7: Extract from Listing 3.3 line 3, 4 and 5

```
1 append b148f67dd8c0081046b196cb5aa8dcc2 == 649b8b471e7d7bc175eec758a7006ac693c434c8297
    ↪ c07db15286788c837154ab148f67dd8c0081046b196cb5aa8dcc2
2 sha256 == bd2c8ac682b8ed4b544ddd29ce229ef42479162b1ff14cdd51c653601600b40b
```

Description: 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.

The result of the concatenated file hash and nonce is then hashed again to produce the value '**bd2c8ac682b8ed4b544ddd29ce229ef42479162b1ff14cdd51c653601600b40b**'

Manual reproduction:

Listing 3.8: Manual reproduction of noncing and hashing

```
1 user@host:~/otsdemo$ python3 unhexSha256Hex.py 649b8b471e7d7bc175eec758a7006ac693c434
    ↪ c8297c07db15286788c837154ab148f67dd8c0081046b196cb5aa8dcc2
2 bd2c8ac682b8ed4b544ddd29ce229ef42479162b1ff14cdd51c653601600b40b
```

unhexSha256Hex.py used in Listing 3.8 is a small Python script that is necessary for manual commitment replays. It is necessary to first convert the textual value back to raw binary format, perform the hashing operation and then convert the hash back to a hexadecimal textual value to be displayed in the timestamp. This is done because the OTS client performs hashing on the raw binary data, which is not printable, and not the textual hexadecimal representation as shown in the commitment operation.

Listing 3.9: Pseudo code for unhexSha256Hex.py

```
1 data = hexToBinary(hexInput)
```

```
2
3 hash = sha256(data)
4
5 hexOutput = binaryToHex(hash)
6
7 print(hexOutput)
```

Step 3 - Noncing per calendar server:

Listing 3.10: Extract from Listing 3.3 line 11 - 16

```
1 -> append 782372bb88a18335daf2a8e596338454 == bd2c8ac682b8ed4b544ddd29ce229ef42479162
      ↪ b1ff14cdd51c653601600b40b782372bb88a18335daf2a8e596338454
2    sha256 == 15bc3053e59447106c4b233c36336389e7e6b5ee7a625121313c7a0b0ebbc75e
3    prepend 59d3cd30 == 59d3cd3015bc3053e59447106c4b233c36336389e7e6b5ee7a625121313c7a0
        ↪ b0ebbc75e
4    append 50d3d70950797d54 == 59d3cd3015bc3053e59447106c4b233c36336389e7e6b5ee7
        ↪ a625121313c7a0b0ebbc75e50d3d70950797d54
5    verify PendingAttestation('https://alice.btc.calendar.opentimestamps.org')
6    sha256 == 56367592cd684c5c2a03e71d353173a921f2441c9ff957cb1bdb926caed295a3
```

Description: Using the output from Step 3, the OTS client performs another round of noncing by appending and prepending values that will serve as indexes to increase the performance of searching for the timestamp in the local or remote OTS cache.

Notice that these indexing nonces are unique per calendar server.

The result of the noncing is then again hashed to produce the start leaf hash (‘**56367592cd684c5c2a03e71d353173a921f2441c9ff957cb1bdb926caed295a3**’) that will be aggregated by the calendar server.

Manual reproduction:

Listing 3.11: Manual reproduction of noncing and hashing and add addition of indexing nonces

```
1 user@host:~/otsdemo$ { echo "bd2c8ac682b8ed4b544ddd29ce229ef42479162b1ff14cdd51
      ↪ c653601600b40b"; echo "782372bb88a18335daf2a8e596338454"; } | tr "\n" " "
2 bd2c8ac682b8ed4b544ddd29ce229ef42479162b1ff14cdd51c653601600b40b782372bb88a18335daf2a8
      ↪ e596338454
3
4 user@host:~/otsdemo$ python3 unhexSha256Hex.py bd2c8ac682b8ed4b544ddd29ce229ef42479162
      ↪ b1ff14cdd51c653601600b40b782372bb88a18335daf2a8e596338454 1
5 15bc3053e59447106c4b233c36336389e7e6b5ee7a625121313c7a0b0ebbc75e
```

```
6
7  user@host:~/otsdemo$ { echo "59d3cd3"; echo "15bc3053e59447106c4b233c36336389e7e6b5ee7
       ↪ a625121313c7a0b0ebbc75e"; } | tr "\n" " "
8  59d3cd315bc3053e59447106c4b233c36336389e7e6b5ee7a625121313c7a0b0ebbc75e
9
10 user@host:~/otsdemo$ { echo "59d3cd315bc3053e59447106c4b233c36336389e7e6b5ee7
       ↪ a625121313c7a0b0ebbc75e"; echo "50d3d70950797d54"; } | tr "\n" " "
11 59d3cd315bc3053e59447106c4b233c36336389e7e6b5ee7a625121313c7a0b0ebbc75e50d3d70950797
       ↪ d54
12
13 user@host:~/otsdemo$ python3 unhexSha256Hex.py 59d3cd3015bc3053e59447106c4b233
       ↪ c36336389e7e6b5ee7a625121313c7a0b0ebbc75e50d3d70950797d54 1
14 56367592cd684c5c2a03e71d353173a921f2441c9ff957cb1bdb926caed295a3
```

Step 4 - Construction of calendar server MHT:

Listing 3.12: Extract from Listing 3.3 line 16 - 38

```
1  sha256 == 56367592cd684c5c2a03e71d353173a921f2441c9ff957cb1bdb926caed295a3
2  append 716f6fe6011712e3556d65265b82eebd545de43b7da9f58530da1e9139251bac == 56367592c
       ↪ d684c5c2a03e71d353173a921f2441c9ff957cb1bdb926caed295a3716f6fe6011712e3556
       ↪ d65265b82eebd545de43b7da9f58530da1e9139251bac
3  ...[29 lines elided]...
4  8d7d1239063b8ba73cee03255f13c19f8ecb6911cb3c9c80de614cf363284ccb45e6f25d147c1d93045
       ↪ f437786e02157cf03d6f63aa8b1a201eeaa8194250fd1
5  sha256 == 37fa6f0f61f4fc61da240549f87f5aff48b2e633e1be9020fbb2b8ecf747bd67
```

Description: The commitment operations depicted in Listing 3.12 are actions that were performed on the calendar server. Specifically, it is representative of the aggregation activities on the calendar server, where the hash is incorporated into a MHT with other submitted hashes as leaves to the MHT.

The more hashes were submitted, and subsequently aggregated by the calendar server in question, the more commitment operations will be in this portion of the timestamp.

Listing 3.12 shows how the output from Step 3, the leaf hash (‘**56367592cd684c5c2a03e71d353173a921f2441c9ff957cb1bdb926caed295a3**’) is concatenated with another leaf hash, and hashed again to produce a new hash that forms the leaf for the next level in the MHT.

This concatenation and hashing is performed for each commitment operation recorded in the timestamp until the MR is produced (‘**37fa6f0f61f4fc61da240549f87f5aff48b2e633e1be9020fbb2b8ecf747bd67**’).

Step 5 - Construct the Bitcoin transaction:

Listing 3.13: Extract from Listing 3.3 lines 39 - 41

```
1 sha256 == 37fa6f0f61f4fc61da240549f87f5aff48b2e633e1be9020fbb2b8ecf747bd67
2 prepend 010000000169845a160e4f3ec0959ac5911a84dea65519713c9ce52bf707197f3ae0ab02
    ↪ e70000000000fdffffff02a0385e0000000000160014e200dde45eb0529aebe86e16060fb9
    ↪ b109008b56000000000000000000226a20 == 010000000169845a160e4f3ec0959ac5911a84de
    ↪ a65519713c9ce52bf707197f3ae0ab02e70000000000fdffffff02a0385e0000000000160014
    ↪ e200dde45eb0529aebe86e16060fb9b109008b56000000000000000000226a2037fa6f0f61f4fc61d
    ↪ a240549f87f5aff48b2e633e1be9020fbb2b8ecf747bd67
3 append e2720700 == 010000000169845a160e4f3ec0959ac5911a84dea65519713c9ce52bf707197f3a
    ↪ e0ab02e70000000000fdffffff02a0385e0000000000160014e200dde45eb0529aebe86e16060f
    ↪ b9b109008b56000000000000000000226a2037fa6f0f61f4fc61da240549f87f5aff48b2e633e1b
    ↪ e9020fbb2b8ecf747bd67e2720700
4 # Bitcoin transaction id 9fd255b44d2373d98382a5469bda07862e1c1f6b89b5a4d7750309d958c
    ↪ e8809
```

Description: Once the MR has been calculated, it serves as input to the Bitcoin transaction. More specifically, the MR will become the OP_RETURN field (the receiving Bitcoin address) of the transaction. Listing 3.13 shows how the construction of the Bitcoin transaction by the Bitcoin client on the calendar server. The structure of a Bitcoin transaction was discussed in an earlier section of this paper.

The important values to note here are the Bitcoin transaction ID (‘**9fd255b44d2373d98382a5469bda07862e1c1f6b89b5a4d7750309d958ce8809**’) and the complete Bitcoin transaction (‘**010000000169...2720700**’) preceding it.

Step 6 - Construction of Bitcoin block transaction MHT:

Listing 3.14: Extract from Listing 3.3 lines 42 - 78

```
1 sha256 == 330ef1825b2394ea48b89e963296e375d139afdaa73f17773ef33f697f2fae09
2 sha256 == 0988ce58d9090375d7a4b5896b1f1c2e8607da9b46a58283d973234db455d29f
3 prepend 85028d0e6bc68dd101f5b3394db660777faa02ef1c7bfdd0440f8ce569e63b4f == 85028d0e6b
    ↪ c68dd101f5b3394db660777faa02ef1c7bfdd0440f8ce569e63b4f0988ce58d9090375d7a4b5896
    ↪ b1f1c2e8607da9b46a58283d973234db455d29f
4 ...[39 lines elided]...
5 d89cf9425ed1905ea9ce2fac1a3256e8cb26de834732070186ec38bf41e59cc237f4d3b525e4b4aaa
    ↪ f28798014e234f4336964dc728e54d6f40718c231439919
6 sha256 == 57414fcb4efefd124f2777f1d07043a97407ca09371cfa98bfcf16834f10f43e
7 sha256 == 34c7c3b00355d8f5fb7dfd9441c3cab033fc94c80442a0466f4143867c1e49a7
8 verify BitcoinBlockHeaderAttestation(488163)
9 # Bitcoin block merkle root a7491e7c8643416f46a04204c894fc33b0cac34194fd7dfbf5d85503b0
    ↪ c3c734
```

Description: As noted in the section on Bitcoin block and transaction structure, the transactions within a Bitcoin block also form a MHT, since each block contains a number of transactions from different sources. Depicted in Listing 3.14 is the process of creating this MHT contained in the Bitcoin block. Note how a double *sha256* hash is now performed as per the Bitcoin specification.

The process starts by hashing the complete transaction from Step 5 ('**010000000169...2720700**') to produce the initial hash in Step 6 ('**330ef1825b2394ea48b89e963296e375d139afdaa73f17773ef33f697f2fae09**'), which is again hashed before being concatenated with another transaction hashes to construct the MHT to arrive at the MR of the block ('**a7491e7c8643416f46a04204c894fc33b0cac34194fd7dfbf5d85503b0c3c734**').

These commitment operations in Listing 3.14 are performed by the Bitcoin network and defined by the Bitcoin protocol in the BTC trust domain at the time of creating the block. These commitment operations are not performed by the OTS calendar server.

Also note the BitcoinBlockHeaderAttestation value **488163**.

Step 7 - Comparison and validation

Description: Now that all of the commitment operations in the timestamp have been replayed from the initial hash of the source file up to the Bitcoin block MR, it is time to look up and validate the data associated with the Bitcoin block and transaction to determine the file integrity and timestamp validity.

The OTS client, as depicted in Configuration C, does this by querying the block in question directly from the local Bitcoin node which would have a copy of all blocks replicated from the Bitcoin network locally. Once the block data is found, the OTS client parses out the block date and time, formats it for easy readability and presents it to the user in their local time zone.

As for the manual proof, a popular web application Blockchain Luxembourg S.A (2017b), as shown in Figure 3.5, that has a block exploration capability is used, allowing a user to search based on criteria, such as block number and transaction ID. Obviously, the OTS client will not perform the same action, as it cannot necessarily trust the data on Blockchain Luxembourg S.A (2017b). In Figure 3.5, a search based on the transaction ID recorded in the timestamp in Listing 3.13 is performed. The transaction data can be located at blockchain.info[1]

[1] `https://blockchain.info/tx/9fd255b44d2373d98382a5469bda07862e1c1f6b89b5a4d7750309d958ce8809`

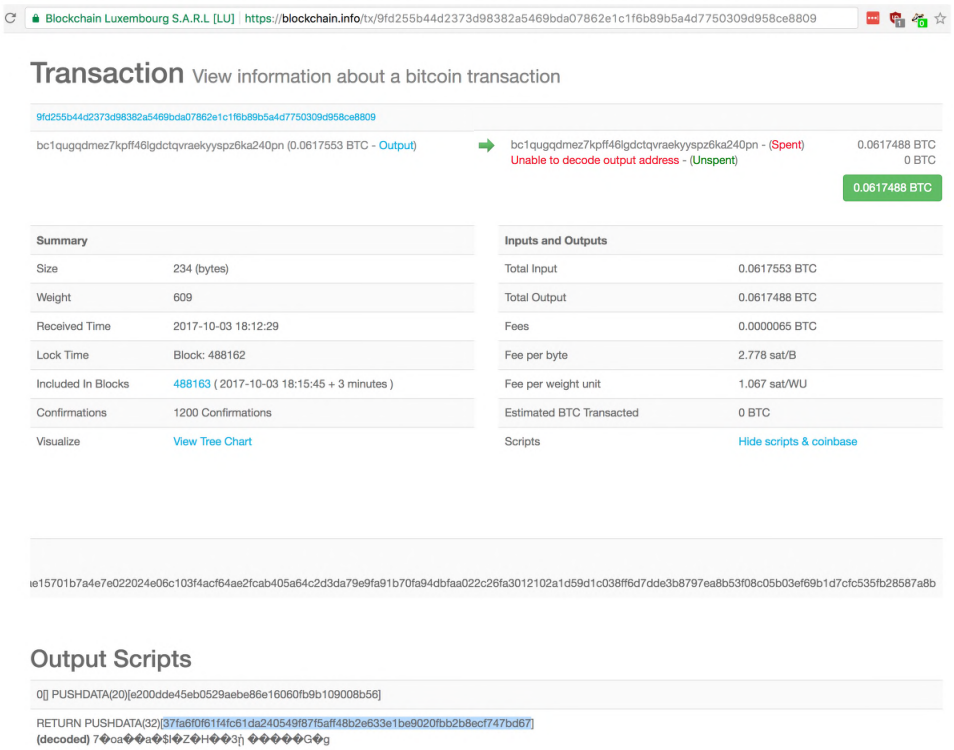


Figure 3.5: blockchain.info lookup of the Bitcoin transaction ID recorded in the timestamp

The result of the search successfully returned a Bitcoin transaction linked to block **488163**. Looking at the scripts associated with this transaction, it can be seen that the root of the MHT calculated as part of the calendar server aggregation in Step 4 (‘**37fa6f0f61f4fc61da240549f87f5aff48b2e633e1be9020fbb2b8ecf747bd67**’) is present in the **Output Scripts** section of Figure 3.5.

Listing 3.15: Transaction output scripts

```
1 RETURN PUSHDATA(32)[37fa6f0f61f4fc61da240549f87f5aff48b2e633e1be9020fbb2b8ecf747bd67]
```

This implies that the original file hash that formed a leaf in the MHT is still that same since the recoded MHT root (‘**37fa6f0f61f4fc61da240549f87f5aff48b2e633e1be9020fbb2b8ecf747bd67**’) and the calculated MHT root (‘**37fa6f0f61f4fc61da240549f87f5aff48b2e633e1be9020fbb2b8ecf747bd67**’) are the same thus proving the integrity of the file.

Furthermore, looking at the block specific data for block **488163**, as shown in Figure 3.6, it can be seen that the block in question has a timestamp date of 2017-10-03 18:15:45. Note that this is UTC time. The block data can be found at `https://blockchain.info/block-index/1627658`

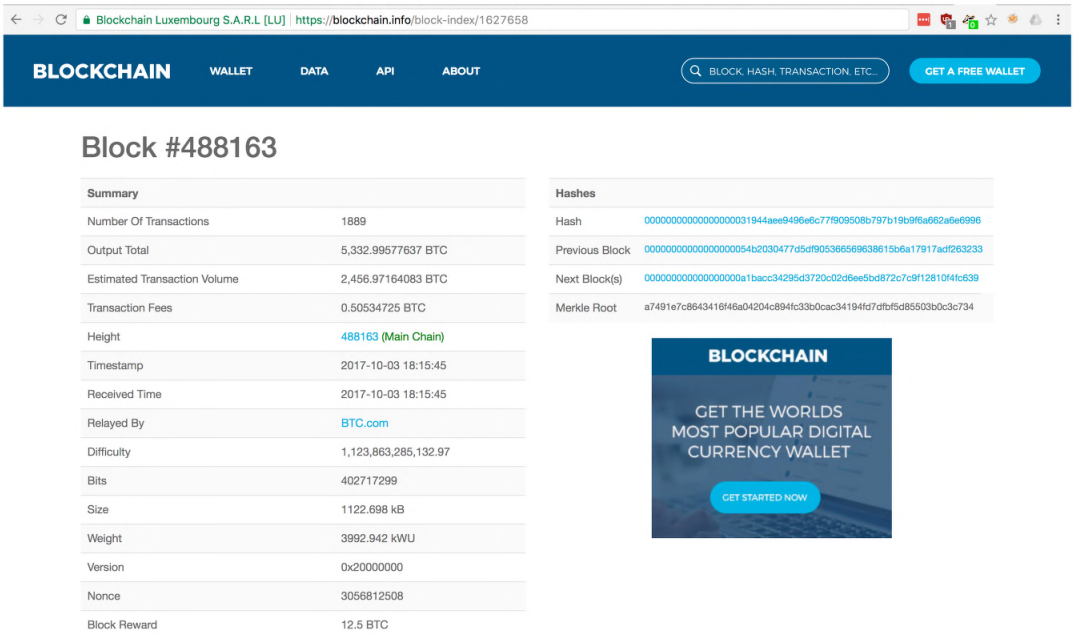


Figure 3.6: blockchain.info lookup of the Bitcoin block number recorded in the timestamp

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 2017-10-03 18:15:45 UTC. Hence the attestation by the OTS client in local time:

Success! Bitcoin attests data existed as of Tue Oct 3 20:15:45 2017 CEST

The complete flow of OTS operations can be seen in Figure 3.7.

3.2.6 OTS challenges, limits and security

Having a much deeper understanding of how OTS works, it is now possible to discuss some of the challenges and limitations faced by the protocol and its implementation as illustrated above in Figure 3.3.

Timing and accuracy

The attestation received by OTS in Listing 3.4 clearly states that the data existed as of a specific time and date, with up to the second accuracy. As Todd (2016b) noted,

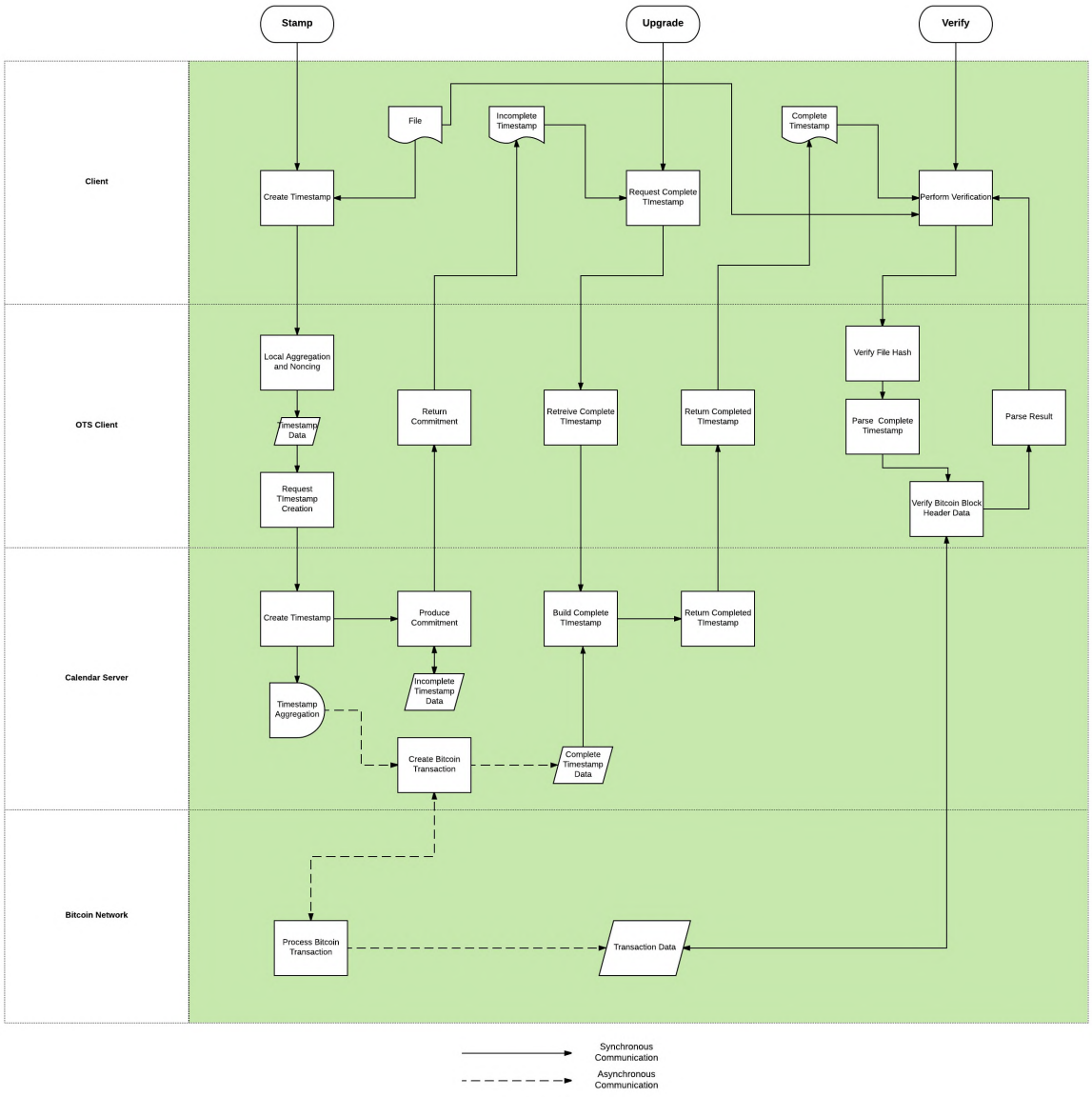


Figure 3.7: Flowchart showing complete OTS operations in Configuration C

this granularity is not necessarily completely accurate up to the second, since the block header time of a Bitcoin block is considered accurate within two to three hours, depending on a range of factors; as he explained: "Every Bitcoin block header has a field in it, called nTime. For a Bitcoin block to be accepted by the network, the Bitcoin protocol requires that the field be set to approximately the time the block was created. Exactly how accurate that field must be for a block to be valid, is a complex topic, but for our purposes it's fair to say it'll very likely be accurate to within two or three hours - even if a sizable minority of Bitcoin miners are trying to create invalid timestamps - and almost certainly within a day."

The details of the accuracy of nTime, and the possibility that it could maliciously be set to an inaccurate time is discussed in detail by Todd (2016a). The possibility of malicious tampering with the nTime field within a block is a function of the number of honest versus dishonest miners that validate the block contents and thus diminishes as the ration of honest to dishonest miners increases. Todd (2016a) noted that if the majority of the hashing power is controlled by dishonest colluding nodes, the situation is hopeless and nTime cannot be trusted, but at that point the entire Bitcoin network can also not be trusted and it would be immediately apparent to the network. Todd (2016a) also noted that inaccuracies in nTime can be accidental and non-malicious in nature, and could be as a result of misinterpretation of daylight savings time or misconfigured NTP servers. In the case of non-malicious nTime however, the probability of correction of the time by any single honest node is very probable and the scope for inaccuracy is reduced to two to three hours.

A secondary aspect of accuracy to consider is the lag introduced by the calendar server and aggregation operations. Because the calendar server aggregates multiple timestamp and incorporates them into a single Bitcoin transaction, there might be considerable lag between when a timestamp is submitted and when it is actually incorporated into a transaction and Bitcoin block. This lag is a side effect of having to rely on a calendar server for performing the timestamp. The frequency with which the calendar server submits aggregated timestamps is up to the administrator of that specific server, but more frequent submissions will result in higher costs, as each individual submission carries a cost.

As mentioned previously, OTS does make provision for use cases where this aggregation lag is not acceptable as it allows the user to either submit directly to the Bitcoin network, if the necessary Bitcoin dependencies are configured locally, or in the form of a local calendar server which can be configured to submit more frequently. In this use case, the

user can configure OTS in such a way as to ensure there is no aggregation which will result in a higher cost.

Clearly the timestamp cannot be accurate up to the second if the accuracy of the block time is not, because the accuracy of the timestamp is directly tied to the accuracy of the block time. Furthermore, the timestamp accuracy can be affected by calendar server aggregation lag which means that a possibly non-negligible amount of time might pass between the submission and incorporation of a timestamp into a Bitcoin transaction by a public calendar server.

We can, therefore, conclude that although the timestamp attests with accuracy up to the second, the timestamp should be interpreted as accurate within two to three hours, and accurate to within a day in a worst case scenario.

The accuracy of OTS timestamps in the configuration depicted in Figure 3.3 will be the subject of further detailed scrutiny in a later section of this work.

Security and privacy

Opentimestamps (2017) notes that privacy and security was a fundamental consideration in the design and implementation of OTS. There are, however, a few trade-offs between usability and privacy as noted earlier when making use of public calendar servers. Even in the case where public calendar servers are used, OTS was designed to protect the confidentiality of the content being timestamped to a very high degree.

This protection comes in the form of noncing, as illustrated in Listing 3.7 and Listing 3.10.

The purpose of this nonce in Listing 3.7 is to add necessary entropy to files of which the entropy is very low. Because there is no authentication on the calendar server, the file hash will be in the public domain and therefore its confidentiality needs to be protected. Smaller files, with low entropy, are susceptible to hash brute force attacks where an attacker that knows the hash can derive the content of the file by brute force guessing the content of the file, hashing it, and comparing it to the known hash. By appending a 128bit nonce the OTS client ensures that the hash that is sent to calendar servers has sufficient entropy to make a brute force attack impractical. The technicalities and effort to brute force attack files with greater than 128bit entropy is outside the scope of this work.

Similarly, the noncing performed in Listing 3.10 further preserves the confidentiality of the submitted hash per calendar server it is submitted to. This per-calendar server noncing ensures that no information that could allow the correlation of interaction with calendar servers is leaked; this is achieved by looking up the same timestamp identifier on multiple servers. If it was possible to gather this sensitive information, an attacker in possession of the timestamp identifier, which is inherently public, would be able to identify which calendar servers are being used by a particular OTS client or set of clients.

Opentimestamps (2017) notes that the biggest privacy risk associated with the use of OTS is the leakage of potentially sensitive metadata: "If you create multiple timestamps in close succession it's quite likely that an adversary will be able to link those timestamps as related simply on the basis of when they were created; if you make use of the timestamp multiple files in one command functionality most of the commitment operations in the timestamps themselves will be identical, providing an adversary very strong evidence that the files were timestamped by the same person.".

Opentimestamps (2017) notes that although the connections to public calendar server is intended to be secured, the calendar server makes no attempt at providing privacy thought mechanisms like authentication or authorisation. Therefore, all calendar server content should be treated as public and the necessary measures taken to preserve the confidentiality of the file content being timestamped. It should be noted that the public calendar server configured by default in the version of OTS being discussed in this work does implement Transport Layer Security (TLS). However, other calendar servers, public or private, might not.

3.3 OTS and Autopsy

For the use and adoption of OTS in the digital forensic discipline to increase, it needs to be implemented in a useful and open way as to allow and encourage scrutiny and further development. OTS has been illustrated to be a comprehensive tool for timestamping and verifying the integrity of digital artefacts, like digital evidence. However, OTS in isolation is less likely to be adopted by digital forensic practitioners if it's not integrated into existing tools and processes. Making the use of OTS in digital forensics a success, depends on how it can be incorporated into existing DF tools for easy adoption.

As noted previously, there is a vast number of both COTS and FOSS forensic tools available to digital forensic practitioners. SleuthKit Autopsy (Autopsy) is but one of

these FOSS tools which has a range of desirable properties that make it a good candidate for integrating OTS. Carrier (2002) noted that SleuthKit is a collection of command line tools with accompanying libraries that enables the user to analyse disk images and recover files from them. Autopsy is an easy to use GUI program that enables the user to efficiently analyse hard drives. Importantly Carrier (2002) noted that Autopsy's plug-in architecture allows the user to find and develop third-party modules that can be integrated into Autopsy to perform a range of tasks. Finally, Carrier (2002) highlighted that Autopsy has thousands of users worldwide and that there is an active community associated with it.

Autopsy, therefore, has three key properties that make it an ideal candidate technology to base further OTS development on:

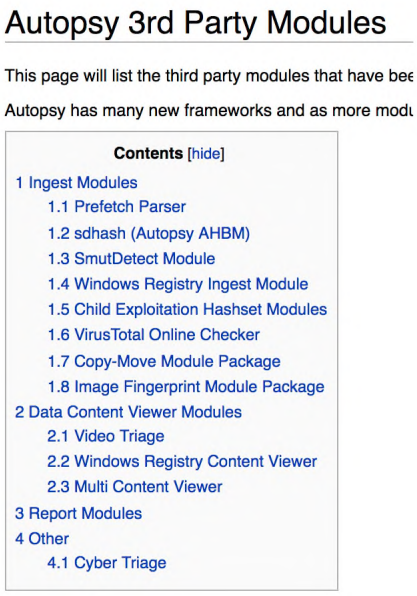
1. Free and open source
2. Plug-in friendly, modular architecture
3. Active user base and community support

Being FOSS means that there is no barrier to using and further developing Autopsy or Autopsy modules. Additionally, it means that Autopsy is open to scrutiny and that any interested party could review and validate its source code.

The modular architecture is another key benefit, as it allows the easy installation and use of any modules developed for Autopsy. The FOSS nature of Autopsy supports this open architecture by allowing potential module developers deeper insight into the core of Autopsy and how to best integrate with it. Being plug-in friendly also implies that developing, installing and using third party plug-ins should be relatively easy.

Lastly, the active user base and community are a good indication that Autopsy has stood the test of time, and will enjoy a growing user base of active user and developers to maintain and progress the platform as requirements and tools advance. This active community ensure that there are many freely available online resources to guide users and developers alike (Sleuthkit.org, 2017b) and (Sleuthkit.org, 2017a).

Sabernick III (2016) also noted that the FOSS nature of Autopsy is the main driving factor behind their use of the framework. Sabernick III (2016) further noted that the ability to develop Autopsy modules, and then share them with the wider community, is one of the most valuable aspects of Autopsy with its open and modular architecture.


Autopsy 3rd Party Modules

This page will list the third party modules that have bee

Autopsy has many new frameworks and as more modu

Contents [hide]

1 Ingest Modules
- 1.1 Prefetch Parser
- 1.2 sdhash (Autopsy AHBM)
- 1.3 SmutDetect Module
- 1.4 Windows Registry Ingest Module
- 1.5 Child Exploitation Hashset Modules
- 1.6 VirusTotal Online Checker
- 1.7 Copy-Move Module Package
- 1.8 Image Fingerprint Module Package

2 Data Content Viewer Modules
- 2.1 Video Triage
- 2.2 Windows Registry Content Viewer
- 2.3 Multi Content Viewer

3 Report Modules

4 Other
- 4.1 Cyber Triage

Figure 3.8: Autopsy wiki listing third-party developed modules

Looking at sources such as Sleuthkit.org (2016), an extract of which is shown in Figure 3.8, certainly reinforces the statement that Autopsy is enjoyed by an active community and that modules are developed for a range of different purposes.

The combination of the above factors make Autopsy the ideal framework for developing an OTS module and exposing OTS to the digital forensics community. The research design will therefore focus on developing an OTS module for Autopsy that would allow users thereof to easily create and verify OTS timestamps for data sources in Autopsy.

Figure 3.9 shows the proposed use case diagram for the Autopsy plugin. The plugin would enable the easy creation and verification of a timestamp for a particular file or set of files by integrating it into the existing software used during such an investigation. Furthermore, the timestamp could also easily be verified outside of the investigator environment as the OTS protocol is open and free; enabling all other parties to verify the timestamp given they have a copy of the file in question and the timestamp created by the investigator. The timestamp can easily be shared with all parties to an investigation for independent verification.

3.4 OTS test design

OTS was analysed in some detail in the above section. The scope of the analysis was narrow and focused on showing how OTS works in a single instance for a specific file.

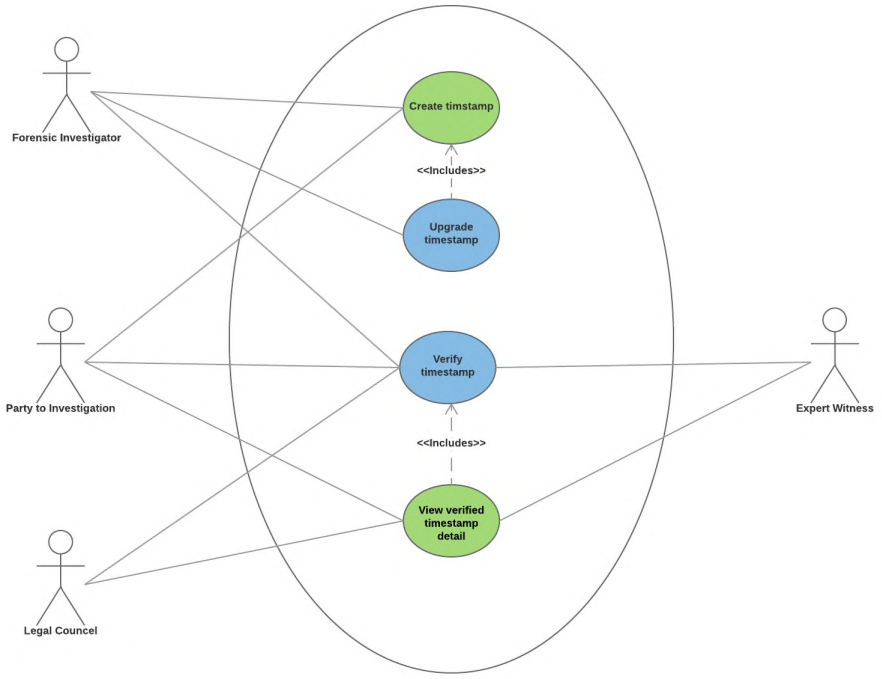


Figure 3.9: Autopsy OTS plugin use case

Given the acquired low level understanding of how OTS creates and verifies timestamps, the analysis is expanded to look at evaluating OTS from a different perspective. To develop a better understanding of how OTS functions at scale, the scope of analysis is shifted from a single timestamp to many timestamps over time. Testing and measurement of OTS at a higher level will allow the determination of its benefits and drawbacks, as well as forming an opinion about its consistency and reliability over a larger sample of files and timestamps.

3.4.1 Design

To perform analyses of the nature described above, the following test design is proposed, namely, one that will continuously and automatically create and validate OTS timestamps, while recording results and key metrics about the process.

Creating and validating OTS timestamps at scale would not be practical if performed manually, as it would take a lot of time and resources. Luckily, the OTS client can easily be invoked programmatically to create and validate timestamps for files.

By automating the OTS functions through the use of a script, a large number of OTS functions to generate large sample of OTS timestamps and record a multitude of data

Table 3.2: OTS test server environment

Virtual Server Details		
Operating System	Name	Ubuntu Xenial
	Version	16.04.3 LTS
	Architecture	x64
CPU		2.2 Ghz (4 Core)
Memory		4096 MB
Storage		200 GB

points about the functions being executed is enabled. The script will be tasked with performing the OTS actions as well as capturing and storing key data points for further analysis.

At a high level the script will perform the following actions:

1. Create a file.
2. Create an OTS timestamp for the file.
3. Verify the OTS timestamp for this file and all previous files not yet verified.
4. Record and store data about items 1, 2 and 3.

Once a large sample of OTS timestamps has been gathered, the associated data set will be analysed to identify any individual or systemic errors with OTS timestamps. Furthermore, the data will be analysed to possibly uncover trends associated with the creation and validation of OTS timestamps; certain performance metrics of OTS will also be examined.

3.4.2 Environment

A stable and consistent environment is essential for this type of testing. The test environment was a dedicated virtual server environment running Ubuntu Linux as well as the necessary software dependencies like Python, OTS and a Bitcoin full node.

Table 3.2 lists the server specifications and operating system.

The core software configuration can be seen in Table 3.3

Table 3.3: OTS test server core software dependencies

Name	Version	Description
Python	3.5.2	Python core libraries
Bitcoin Core	v0.15.0.0-g3751912e8e	Bitcoin core libraries including the bitcoind daemon and the bitcoin-cli
OpenTimestamps Client	v0.5.0	The OpenTimestamps client libraries.
MongoDB	v3.2.17	MongoDB libraries including the mongod daemon

All of the software dependencies were installed with default configurations where possible. Each software package listed in Table 3.3 installed and configured their own dependencies as per the normal install process. These sub-dependencies are not within the scope of this discussion.

Python 3+ is required for the OTS client, which will also be the programming language for the test script that would interact with the OTS client and the database.

Bitcoin Core libraries are necessary to run a local Bitcoin node to interact with the Bitcoin network. In the testing configuration, Bitcoin would not function as a wallet, but only a full node that maintains a copy of the Bitcoin blockchain. From an OTS perspective, the Bitcoin libraries would only be used to verify timestamps and not to create them; calendar servers will still be used to create these timestamps. The Bitcoin node was set up according to guidelines by Bitcoin Project (2015)

The OTS client, necessary to perform all the OTS functions, was installed and configured according to the project documentation found at Opentimestamps (2017).

Finally, MongoDB, serving as the data store for the recorded OTS function data, is required. MongoDB was installed according to the product documentation noted by MongoDB Inc. (2016).

3.4.3 Test script

The test script called *ots-test.py* is the main component of the OTS tests and is responsible for all OTS functions, execution, and data gathering. The script code can be seen in Listing A.1. Listing 3.16 is the pseudocode for *ots-test.py* and describes at a high level what the script does and in which sequence.

Listing 3.16: ots-test.py pseudocode

```
1  define otsobj
2
3  CreateAndStampNewFile():
4    create NewFile with random contents of random size between 1 and 1024 bits in Workin
       ↪ gDirectory
5
6    otsStamp NewFile
7
8    populate otsobj with otsStamp results
9
10   save otsobj to Database
11
12 UpgradeExistingTimestamps():
13   OtsObjCollection = query all otsobj from DB that are not yet complete timestamps
14
15   for otsobj in OtsObjCollection:
16     otsUpgrade ExistingFile
17
18     update otsobj with otsUpgrade results
19
20     save otsobj to Database
21
22 VerifyCompleteTimestamps():
23   OtsObjCollection = query all otsobj from DB that are complete and not yet verified
24
25   for otsobj in OtsObjCollection:
26     otsVerify ExistingFile
27
28     update otsobj with otsVerify results
29
30     save otsobj to Database
```

As is clear from Listing 3.16, the sequence of actions in *ots-test.py* is very straight forward: Create a new file and timestamp it; then Upgrade all existing incomplete timestamps; and finally, verify all unverified complete timestamps. During each of these operations, the necessary information and results are saved to *otsobj*, which in turn is saved to the database for persistent storage.

Script structure and data

The structure of *otsobj* warrants further discussion. The data structure in question is created shown Listing 3.17

Listing 3.17: otsobj data structure

```
otsobj = {
  'name': '',
  'path': '',
  'proof': '',
  'events': [],
  'size': ''
}

otsevent = {
  'name': '',
  'time': '',
  'command': '',
  'message': ''
}

otsproof = {
  'created': False,
  'committed': False,
  'info': '',
  'upgraded': False,
  'verified': False
}
```

OtsObj is a nested object used to store the file and timestamp data for each stamped file created by the script. The object has a range of simple properties, such as *name*, *path* and *size*, combined with complex properties, which themselves are collections like *proof* and *events*. The final structure of the object with some reference data is shown in Listing 3.18.

Listing 3.18: Populated otsobj

```
{
    "_id" : ObjectId("59af06f3b88c36077903107a"),
    "name" : "1504642801.746642.tst",
    "proof" : {
         "verifiedTime" : 1504643433,
```

```
6       "info" : "Assuming target filename is '/home/user/ots-tests/testfile
            ↪ s/1504642801.746642.tst'\nSuccess! Bitcoin attests data existed as of Tu
            ↪ e Sep 5 22:30:33 2017 CEST",
7       "verified" : true,
8       "created" : true,
9       "committed" : true,
10      "attestationDetail" : "File sha256 hash: 14dcaeadfafc615eb32f13fe7e2f09d8480
            ↪ a5558d6798b0335b5851fb3408cd9[ELIDED PROOF]",
11      "upgraded" : true,
12      "createdTime" : 1504642803.4072738
13  },
14  "events" : [
15      {
16          "command" : "create",
17          "time" : 1504642801.747986,
18          "message" : "Created new file /home/user/ots-tests/testfile
                ↪ s/1504642801.746642.tst",
19          "name" : "CreateFile"
20      },
21      {
22          "command" : "stamp",
23          "time" : 1504642803.5810077,
24          "message" : "Stamped file /home/user/ots-tests/testfiles/1504642801.746642.
                ↪ tst with result True",
25          "name" : "StampFile"
26      },
27      {
28          "name" : "UpgradeFile",
29          "time" : 1504644604.9855325,
30          "message" : "Performed an upgrade and the result was True",
31          "command" : "upgrade"
32      },
33      {
34          "name" : "VerifyFile",
35          "time" : 1504644609.7777689,
36          "message" : "Performed a verification and the result was Assuming target fi
                ↪ lename is '/home/user/ots-tests/testfiles/1504642801.746642.tst'\nSuc
                ↪ cess! Bitcoin attests data existed as of Tue Sep 5 22:30:33 2017 CEST
                ↪ ",
37          "command" : "verify"
38      }
39  ],
40  "size" : 428,
41  "path" : "/home/user/ots-tests/testfiles/1504642801.746642.tst"
```

```
42 }
```

The object mentioned in Listing 3.18 is created for each file by the script at the time of creating the file and stamping it. This serves as the data structure in which the data about the timestamp and associated functions will be recoded. The object is saved to the database and subsequently read from the database whenever the file or timestamps is operated on. Any changes are recorded in the appropriate field within the data structure and are subsequently written to the database for persistent storage.

The base properties of *OtsObj* are:

1. name: The name of the created file.
2. proof: A nested complex object that contains data about the timestamp status.
3. events: A collection of nested complex objects that contain all events related to the timestamp.
4. size: The size (in bytes) of the created file.
5. path: The complete path to the created file.

All time fields are recorded as UNIX time (UT), or seconds since 00:00:00 Jan 1 1970 UTC, to avoid ambiguity and complexities with time zones and conversions.

Many fields are metadata fields to help keep track of the status of the file and its progress through the various OTS operations, and are not necessarily significant to the overall testing goal. They will therefore not be discussed in detail. Table 3.4 gives a brief overview of the most important fields recorded by the script and their meaning.

The above mentioned fields, as well as other metadata about the associated files, are recorded and maintained for each file and related timestamp as each OTS function is executed. As soon a particular timestamp has a *proof.verified* value of *True* the timestamp is considered complete and no further actions are taken or recorded for that timestamp.

Script timing and execution

The script was set to execute automatically on the Ubuntu server by configuring a *systemd* job. The timing of the execution was set to every ten minutes. This timing was chosen

Table 3.4: Notable fields present in OtsObj and descriptions

OtsObj fields		
Name	**Data format**	**Description**
name	text	The name of the created file as UNIX time of the creation time appended with the .tst (test) extension.
proof.verifiedTime	UNIX time	The time reflected in the timestamp verification. The time at which OTS attests the data existed.
proof.info	text	The informational message message returned by the OTS client. The output of the OTS client as if invoked through the CLI.
proof.verified	boolean	Indicator if the timestamp has been verified by OTS. OTS Verify function was successful.
proof.created	boolean	Indicator if the timestamp was created. OTS stamps command was executed for the file.
proof.committed	boolean	Indicator if the timestamp was submitted to the calendar server. OTS stamps command was successful and retuned a valid result from the calendar server.
proof.upgraded	boolean	Indicator if the timestamp was upgraded. OTS Upgrade function was executed and retrieved a complete timestamp from the calendar server.
proof.createdTime	UNIX time	The time the proof.committed result was returned. The time the committed timestamp was confirmed.
events[x].name	text	The name of the event. Describes the nature of the action that triggered the event.
events[x].time	UNIX time	The time the event occurred.
events[x].message	text	The description and/or output from the event.

for two reasons. Firstly, to ensure that a large sample of timestamps could be generated in a reasonable amount of time (at least 3 000 in two weeks of continuous execution). Secondly, to ensure that there is a large enough time gap between executions to enable the script to perform all of the necessary OTS functions, thereby preventing concurrent execution that could lead to data integrity issues.

By running the above-mentioned script at a regular and constant interval, a significantly sized data set of timestamp data can be gathered, with sufficient granular data points to analyse OTS trends and potential error rates. The data was also recoded in a structured and easy-to-consume manner which enables easy retrieval and analysis. Apart from the data set mentioned above, all files and associated timestamps are saved to perform spot checks and verification on, if necessary. Finally, some metadata about the script execution itself is logged and recorded to enable troubleshooting or log correlation if necessary.

3.5 Summary

In this section, the OTS protocol was discussed and investigated in detail to understand how and why it can potentially be trusted. Multiple configuration options were discussed, along with how each configuration results in a unique balance between convenience and security. A better understanding of its potential strengths and weaknesses is understood and, through a practical example, it was shown how the verification mechanism works in the desired configuration for an optimal balance of usability and security.

Subsequent to the discussion of OTS, Autopsy as a vehicle for OTS integration was looked at and it was concluded that Autopsy is the ideal platform as a result of its open nature and modular design. Autopsy also has an active community of support and development resources to enable the development of an Autopsy OTS module. This subsection was concluded with a basic Autopsy OTS use case and a brief discussion on how different parties to an investigation could use it.

Finally, a design to test OTS at scale to investigate factors such as usability, error rates, and response times was discussed and investigated. A verbose set of metrics was recorded with the creation of a large sample of timestamps to generate a data set of OTS-related data to investigate further and draw potential insights.

Chapter 4

Implementation

4.1 Introduction

In this section of the paper, a discussion on how the OTS protocol was implemented within the Autopsy framework to create an easy-to-install and use Autopsy module is presented. The module allows a user of Autopsy to timestamp data sources as they are imported into an Autopsy case (project) and view the timestamp results in an easy-to-understand format. A brief overview of how Autopsy module development works, the dependencies and the development environment are outlined. The autopsy-opentimestamps module and its functions are discussed, and its functions briefly illustrated. The section is concluded with a discussion about some challenges experienced during development and how the results are believed to be useful. This is followed by a discussion of the end result, and how and where the module can be downloaded.

4.1.1 Autopsy module development

As noted in the previous section, Autopsy is an application and framework that was built with modularity and extensibility in mind. It is, therefore, quite easy to find resources on developing modules for the platform as is apparent in Sleuthkit.org (2015). The Autopsy Developer's Guide notes: "Autopsy was developed to be a platform for plug-in modules.

The Developer's Guide contains the API docs and information on how to write modules. When you create a module, add it to the list of Autopsy 3rd Party Modules."

It further clearly states that there are two types of Autopsy development:

1. Development of core Autopsy infrastructure, including the Autopsy framework
2. Development of modules that can be used with the Autopsy framework.

The focus of this work is on the second type of development, where a module is developed that can be freely distributed for use by the community. Sleuthkit.org (2017a) noted that modules can be developed in either Python or Java. Autopsy itself is developed in Java but can incorporate modules developed in Java using Jython, a free and open Java implementation of Python.

As noted in Sleuthkit.org (2017a), one of the main motivations to develop an Autopsy module rather than a standalone tool is that Autopsy abstracts a lot of the laborious management functions like data inputs, outputs and presentation. This enables the module to focus primarily on the analysis of data.

From a developer's perspective there are a few basic terms that should be well understood to start developing an Autopsy module. Firstly, a case, which translates broadly to an Autopsy project and is a logical container for data sources and resources related to that project. A case can have many data sources. Secondly, there are data sources which refer to disk images or collections of logical files. The central database is another important component and serves as the persistent storage layer for modules to write and read metadata and analysis results. The Blackboard is a form of intermediate storage shared between modules and can be used to communicate data between two or more modules by posting what is called artefacts to the Blackboard. Finally, there are many services and utilities available to developers as depicted in the list below.

These services and utilities are primarily exposed as APIs and provide supporting functions to the module. These include:

1. File Manager: An API to interact with files in a Case.
2. Logging: API for logging informational or error messages to the Autopsy log file.
3. Pop-up windows: For providing user feedback in the GUI.

4. Module settings: Used for storing persistent configuration between invocations of Autopsy.
5. Content Utilities: A collection of utility methods to write files to local disk.
6. Platform Utilities: API to determine user context and save resources to the user directories.
7. File Utilities: Utility for manipulating folders.
8. Ingest Services: Exposes a collection of generic ingest services.

Using a collection of the above-mentioned services and resources modules can be developed with rich functionality and standardised interaction with the framework and underlying data storage mechanisms.

There are four main types of Autopsy modules that can be leveraged by modules developers to perform different functions.

1. Ingest modules
2. Report modules
3. Content viewers
4. Result viewers

Ingest modules, as the name suggests, are used to operate on data as it is ingested into an Autopsy case. Ingestion can happen when the case is created initially or throughout the lifecycle of the case as more data sources are added. Ingestion modules can be further classified into two types, file ingest modules and data source ingest modules. File ingest modules are triggered for each individual file in a data source whereas data source modules are triggered once per data source.

Report modules are usually invoked after ingestion and are used to deliver the results of any analysis to the user. Report modules can also be used to perform further analysis if desired.

Content Viewers are modules with graphical components and allow a module to display content of a data source in a specific and visually appealing way; I.E., graphs, rendered images or plain text.

Table 4.1: Development workstation configuration

Development Workstation Details		
Operating System	Name	Windows
	Version	10.0.15063
	Architecture	x64
CPU		2 Ghz (4 Core)
Memory		4096 MB
Storage		90 GB

Table 4.2: Development environment software dependencies

Name	**Version**	**Description**
NetBeans IDE	8.2 Patch 2	Netbeans IDE for developing NetBeans modules
JRE	8.0.1440	Java runtime environment
JDK	8.0.1440	Java development kit
Bitcoin	v0.14.2	Bitcoin core libraries including the bitcoind daemon and the bitcoin-cli
Autopsy	4.4.0	Autopsy framework and application
java-opentimestamps	v1.14	Java implementation of OTS

Result Viewers are modules to present data and information related to a collection of files in a data source or case.

This brief overview of Autopsy module development resources and the architecture of the Autopsy framework clearly illustrate that the platform is ideally suited to support and promote easy module development. Additionally, the framework and its services are well documented and present a rich set of standardised functionality.

4.1.2 Development environment

For the development of the autopsy-opentimestamps module, a dedicated, Integrated Development Environment (IDE) was configured on a virtual workstation. The IDE had the high level configuration shown in Table 4.1

Furthermore, the IDE had the primary software dependencies shown in Table 4.2. The choice of operating system and software dependencies present in the development environment were driven by some compatibility prerequisites. Autopsy 4.4.0 is fully supported only on Windows and thus an IDE running on Windows was necessary. Autopsy modules also require NetBeans as Autopsy modules are built on top of the NetBeans Rich Client platform to support the plug-and-play nature of Autopsy modules. Guidance on

Sleuthkit.org (2015) was followed to install and configure NetBeans and to create a basic module project.

The Java Runtime Environment (JRE) and Java Development Kit (JDK) listed in Table 4.2 are required by both NetBeans and Autopsy. As with the OTS testing script, a local Bitcoin node was required and was installed to enable OTS to interact with the Bitcoin blockchain. The choice of java-opentimestamps as opposed to the Python variant, used for the OTS test script, was twofold. Firstly, although Autopsy can support modules written in Python, its use of Jython means that it can only support modules written in Python 2.7. OTS however, requires, at a minimum, Python 3.5 to run. This would mean that, in order to use the OTS libraries from the Python implementation, they would have to ported to Python 2.7. This in turn would require significant effort outside the immediate goals of this project. Luckily, a Java implementation of OTS was also available and it was decided that this OTS implementation would be used for the development of the autopsy-opentimestamps module. Secondly, Autopsy itself is written in Java and thus natively supports modules developed in Java, resulting in less complexity.

All of software dependencies listed in Table 4.2 were installed using the default configuration options.

4.1.3 Module design

During the initial design phase it was important to make the appropriate design decisions to ensure the module would be functional. As discussed in the previous section, there are multiple types of modules within the Autopsy framework and deciding which would be most appropriate for the development of an OTS module would be crucial. Since the purpose of the autopsy-opentimestamps module is to create and validate timestamps for data sources it was determined that the most appropriate module type was an ingest module.

By using an ingest module design, the module could create timestamps for data sources as soon as they are ingested into a case. This default behaviour of timestamping at the time of ingestion would mean that the possibility of accidentally not timestamping a data source would be minimised.

A secondary design consideration involved the type of ingest module. As discussed, an ingest module can be either a file ingest module or a data source ingest module. It was

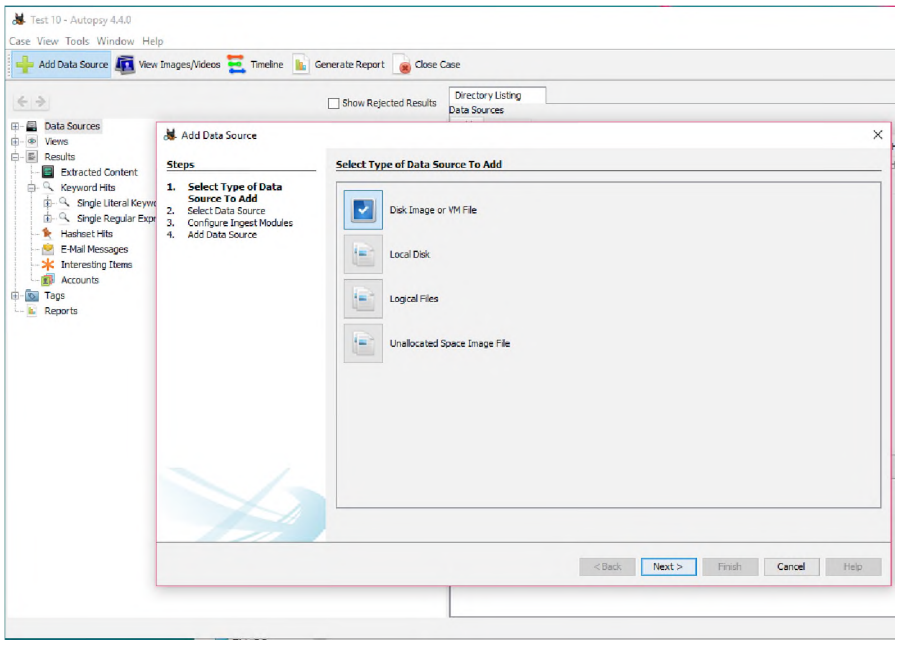


Figure 4.1: Autopsy 'Add Data Source' dialogue

decided that a data source ingest module would be best suited to achieve the desired functionality. This would mean that timestamps would be created at a data source level at not necessarily at an individual file level. Autopsy supports a range of data sources as can be seen in Figure 4.1.

Each of these data sources possess a unique set of characteristics that dictate how they can be used within an ingest module. As an example the "Disk Image or VM File", when ingested, is seen as single container for a collection of other files whereas the "Logical Files" data source type is not seen as a container but rather a collection of individual files. This meant the module would have to process these data sources differently.

In the case of a disk image, which itself is a container for a hierarchical file structure, it would be sufficient to timestamp only the container since its hash would by design include all the files within the container. This would imply that if any single file or artefact within that container changed, the hash of the container would also change and invalidate the timestamp, as expected. With a disk image a file or folder contained within its structure cannot be moved outside of the container without altering the hash of the container. By having a logical container, such as a disk image, it would not be necessary to timestamp each file within that container.

Conversely, the logical file set data source does not necessarily belong to a container with a hierarchical file structure. This means there is no top-level element that can be

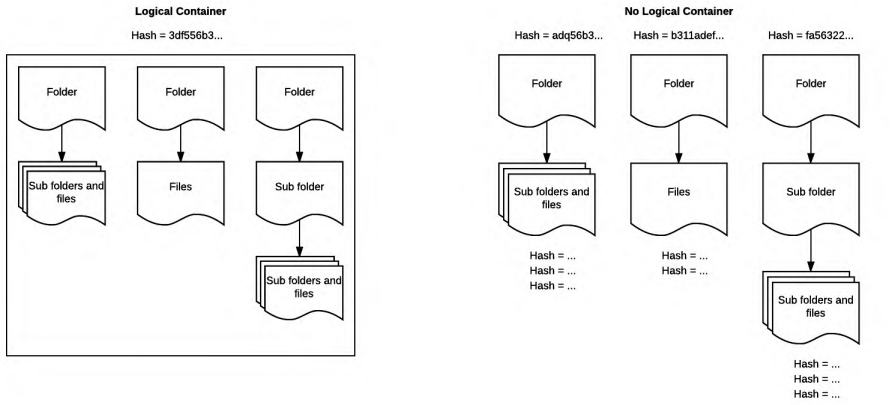


Figure 4.2: Data source logical structures

timestamped that would include all of the files within the data source. In the case of a logical file set where multiple folders or files can be at the same level, each of the files in the file set would have to be individually timestamped. This is achieved within an ingest module by recursively enumerating each of the folders and timestamping all files at that level. Essentially the behaviour of a file ingest module in mimicked within a data source ingest module to achieve the desired functionality.

Figure 4.3 shows the high-level logical execution flow of the autopsy-opentimestamps module for all supported data source types.

Note that there are two distinct sources of execution: execution upon ingestion and manual execution. Both of these execution paths would operate on the selected data source. In both cases, when importing a new data source or manually running ingest modules from the context menu, a user will be presented with the screen shown in Figure 4.4 whereby they can choose to list private calendar servers if they wish to use those. By leaving this options empty the module will use the default calendar servers configured in OTS. Support for extended configuration options like proxy awareness and custom Bitcoin node configuration shown in Listing 3.1 was specifically excluded from the scope of this module implementation since the aim was to create a minimum viable solution which is both easy to use and simple configure.

When invoked, the first action by the module is to determine the data source type, since different data sources need to be handled differently. If the data source is an image file, the module proceeds to check if a timestamp already exists for that data source. If the data source is a set of logical files, it enumerates each of the files and, in turn, checks if a timestamp exists for the file in question.

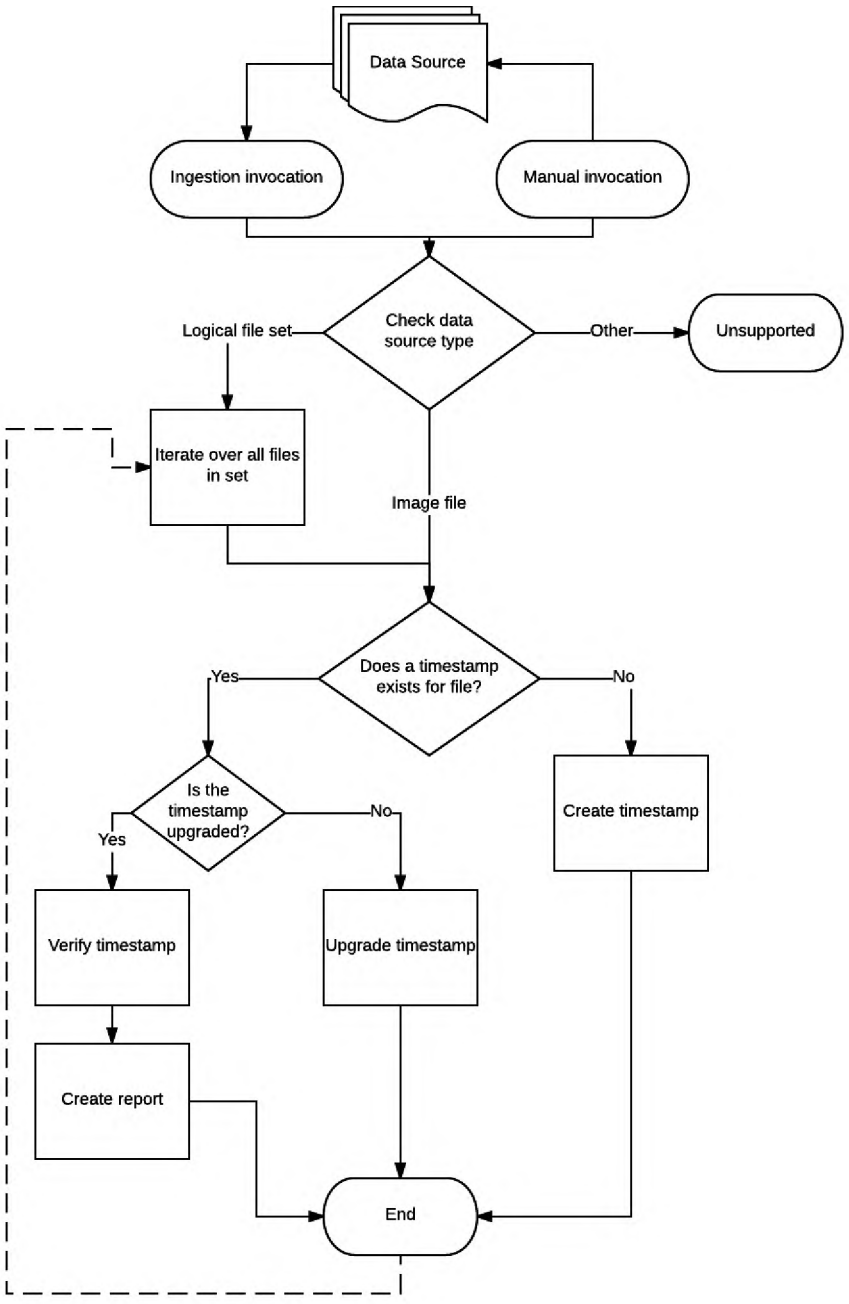


Figure 4.3: autopsy-opentimestamps module execution flow for data sources

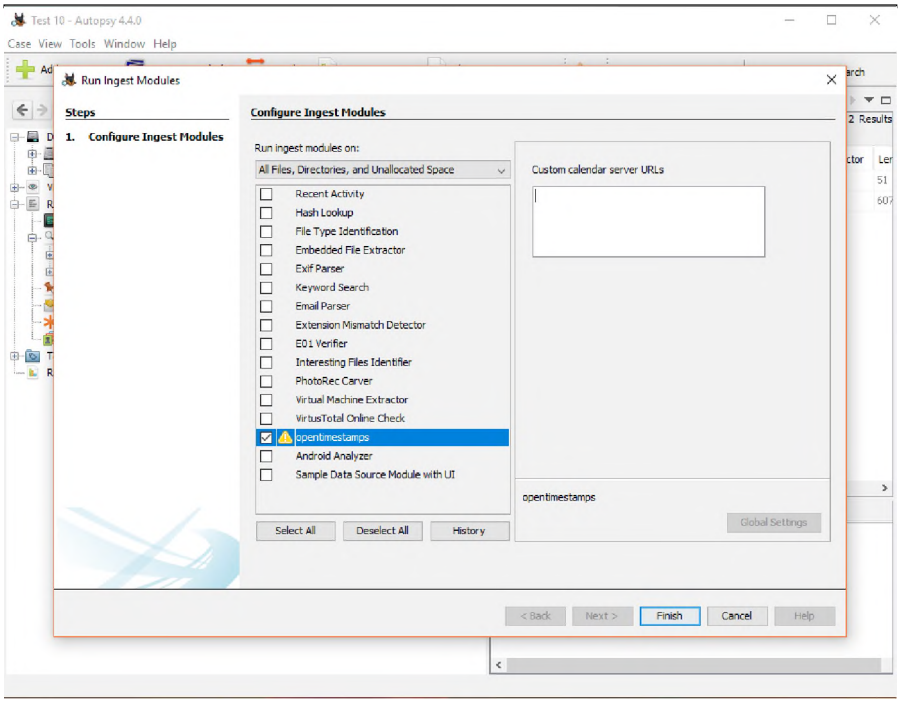


Figure 4.4: autopsy-opentimestamps options screen

If a timestamp for the file does not exist, the module proceeds to create a timestamp and exits or returns control to the enumeration process to advance the next available file. If a timestamp does exist for a file, the module checks if the timestamp is complete. In the case of an incomplete timestamp, the module attempts an Upgrade to retrieve the complete timestamp from the calendar server and exits the current execution. If the timestamp is already upgraded, the module performs a verification operation and logs the results to a report which it saves locally before exiting. All actions performed on each of the files are logged to the Autopsy log file for audit purposes. A sample of such a log file can be seen in Figure 4.5.

The results of OTS operations are recorded in an OTS report for each data source in a case. The reports are displayed in the dedicated Autopsy report section in the left-hand pane of the Autopsy user interface, as can be seen in Figure 4.6. The report can be viewed by clicking on the report entry, which will open the report in the containing folder.

The OTS report is text based, and the name is derived from the associated data source name, appended with the text *_OTS_Report.txt*.

An example of the content of the OTS report can be seen in Figure 4.7, which notes all the important OTS operations, such as timestamp creation, timestamp Upgrade results, as well as verification results. This report is append-only, and any subsequent verification

```
97  Sun Sep 10 21:03:49 GST 2017 org.sleuthkit.autopsy.modules.opentimestamps.OpentimestampsModule getDataSourcePaths
98  INFO: Found file in logical file set - this is the path: E:\TestDataSources\Case 10\bundeled-nested\ts5.txt
99  Sun Sep 10 21:03:49 GST 2017 org.sleuthkit.autopsy.modules.opentimestamps.OpentimestampsModule process
100 INFO: We have more than one path
101 Sun Sep 10 21:03:49 GST 2017 org.sleuthkit.autopsy.modules.opentimestamps.OpentimestampsModule checkOtsProofExists
102 INFO: Checking if E:\TestDataSources\Case 10\bundeled-nested\bundeled\ts1.txt.ots exists.
103 Sun Sep 10 21:03:49 GST 2017 org.sleuthkit.autopsy.modules.opentimestamps.OpentimestampsModule checkOtsProofExists
104 INFO: E:\TestDataSources\Case 10\bundeled-nested\bundeled\ts1.txt.ots does exist.
105 Sun Sep 10 21:03:49 GST 2017 org.sleuthkit.autopsy.modules.opentimestamps.OpentimestampsModule otsProcess
106 INFO: OTS process - Proof exists so we will attmept to upgarde it
107 Sun Sep 10 21:03:49 GST 2017 org.sleuthkit.autopsy.ingest.IngestManager startIngestJob
108 INFO: Ingest job 0 started
109 Sun Sep 10 21:03:51 GST 2017 org.sleuthkit.autopsy.modules.opentimestamps.OpentimestampsModule upgradeOtsProof
110 INFO: Timestamp successfully upgraded
111 Sun Sep 10 21:03:51 GST 2017 org.sleuthkit.autopsy.modules.opentimestamps.OpentimestampsModule createOtsReport
112 INFO: About to writer report, path is: E:\Autopsy\Autopsy Cases\Test Case 10\Test 10\ModuleOutput\opentimestamps\LogicalFileSet1_OTS_Report.txt
113 Sun Sep 10 21:03:51 GST 2017 org.sleuthkit.autopsy.modules.opentimestamps.OpentimestampsModule createOtsReport
114 INFO: Added line to report: 2017/09/10 21:03:51: Timestamp successfully upgraded
115 Sun Sep 10 21:03:51 GST 2017 org.sleuthkit.autopsy.modules.opentimestamps.OpentimestampsModule otsProcess
116 INFO: OTS process - It was upgraded so we'll just move along and verify it
117 Sun Sep 10 21:03:51 GST 2017 org.sleuthkit.autopsy.modules.opentimestamps.OpentimestampsModule verifyOtsProof
118 INFO: E:\TestDataSources\Case 10\bundeled-nested\bundeled\ts1.txt.ots
119 Sun Sep 10 21:03:55 GST 2017 org.sleuthkit.autopsy.modules.opentimestamps.OpentimestampsModule verifyOtsProof
120 INFO: Success! Bitcoin attests data existed as of Sat Sep 09 20:57:14 GST 2017
121 Sun Sep 10 21:03:55 GST 2017 org.sleuthkit.autopsy.modules.opentimestamps.OpentimestampsModule createOtsReport
122 INFO: About to writer report, path is: E:\Autopsy\Autopsy Cases\Test Case 10\Test 10\ModuleOutput\opentimestamps\LogicalFileSet1_OTS_Report.txt
123 Sun Sep 10 21:03:55 GST 2017 org.sleuthkit.autopsy.modules.opentimestamps.OpentimestampsModule createOtsReport
124 INFO: Added line to report: 2017/09/10 21:03:55: Success! Bitcoin attests data existed as of Sat Sep 09 20:57:14 GST 2017
125 Sun Sep 10 21:03:55 GST 2017 org.sleuthkit.autopsy.modules.opentimestamps.OpentimestampsModule createOtsReport
126 INFO: Added line to report: 2017/09/10 21:03:55: Timestamp details: File sha256 hash: 430874f69f7c17e24ecbd9c6aa00b063621fbb3d47d24e52614fd42f0affe84b
```

Figure 4.5: Autopsy log file showing autopsy-opentimestamps operations

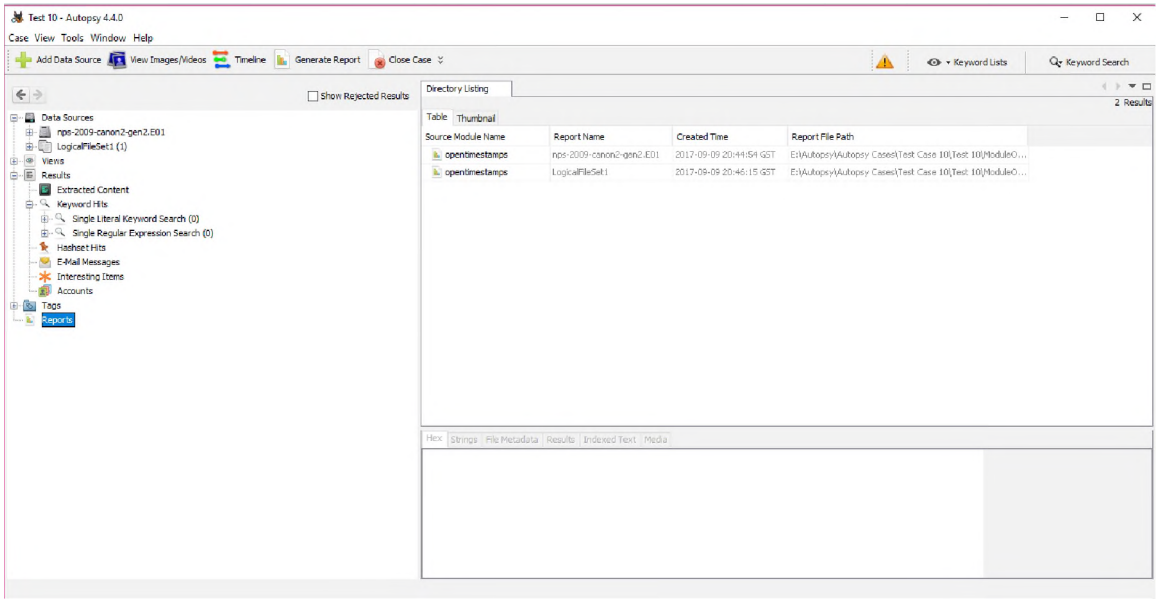


Figure 4.6: Autopsy report viewer

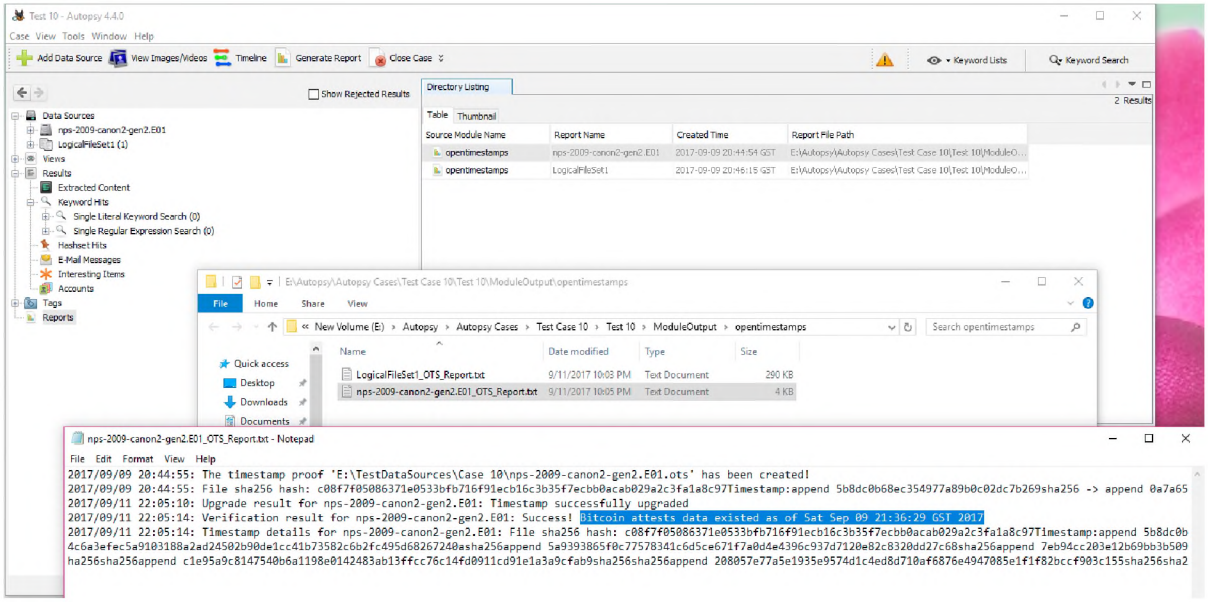


Figure 4.7: Autopsy report viewer and OTS report example

results will be appended to the end of the report for that data source. The report also contains the complete list of OTS timestamp commitment operations to enable a manual proof, if necessary.

As is clear from Figure 4.8, there are a number of source code components to the autopsy-openstimestamps project under the source packages view. Many of these components, for instance OpentimestampsModuleFactory.java and OpentimestampsJobSettingsPanel.java, are necessary module components for any Autopsy module and are created in accordance with the guidance on Sleuthkit.org (2015). The most relevant modules to the OTS functionality are OpentimestampsModule.java and OpentimestampsFunctions.java, which will be briefly discussed below.

OpentimestampsModule.java is the main component which directs the flow of execution shown in Figure 4.3. This Java class is responsible for all the execution logic, reporting logic, and logging. It is initialised with the relevant module settings when the module is invoked and will process the data source being passed to it. This class has a number of functions that in turn call the OpentimestampsFunctions.java class for OTS operations.

OpentimestampsFunctions.java is a wrapper class for the java-opentimestamps project which extends OTS methods exposed by OtsCli.jar. Note the OtsCli.jar dependency in the Libraries section in Figure 4.8. The java-opentimestamps project compiles to OtsCli.jar, which exposes OTS methods such as Stamp, Upgrade, and Verify in a CLI-friendly manner. For use outside of a CLI environment, the methods exposed by OtsCli.jar

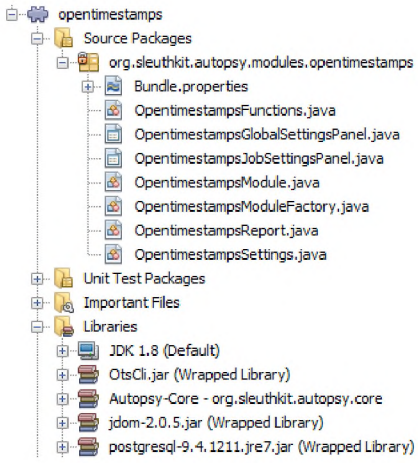


Figure 4.8: autopsy-opentimestamps project structure

are not ideal, as they take text-based input parameters and provide textual return types. Because of this, these methods were reimplemented in OpentimestampsFunctions.java with defined data types as input parameters, and with complex return types such as lists. These complex return types made it easier to programmatically interact with the OTS methods by passing more complex objects to and from OTS calling methods in OpentimestampsModule.java.

4.2 Challenges

During the implementation phase it was confirmed that the Autopsy framework does provide a rich feature set to simplify the development of modules. It was also observed that the guides, sample projects, and how-to modules listed on Sleuthkit.org (2015) were very useful to guide development.

Implementation did, however, become more difficult when attempting to build functionality, not implemented in an example project. It was also unclear how to approach development where a module should act as both a data source ingest module and a file ingest module. This lack of clarity led to the development of custom logic to determine the data source type and to choose the appropriate execution path.

Implementation was started without having a clear idea of how the OTS result would be displayed to the user, and several options were explored during development. Initially, the Blackboard seemed like the appropriate solution due to the ease of access, but it was found that artefacts posted to the Blackboard are stored in volatile memory and would

not persist across multiple executions. This led to the use of the Report functionality rather than the Blackboard.

It was found that module execution was primarily geared towards synchronous execution and feedback, and proved to be difficult to design a feedback mechanism that could persistently show the state of a long-running asynchronous operation (like OTS upgrade).

Of particular bearing is that a defect in java-opentimestamps was discovered during implementation; it was reported, a fix proposed and accepted shortly thereafter. The fix can be seen in the java-opentimstamp project code repository on Github[1]. The concurrency error was detected when the autopsy-opentimestamps module attempted to successively upgrade timestamps for a large logical file set in quick succession.

Finally, it was found that the ability to give real time rich feedback to the user, although possible, was not easy given the functionality in the Autopsy framework. Even though there is a feedback mechanism during module execution, it is limited to a progress bar.

In order to support testing and validation of the module by the community, it was decided to use publicly accessible data sources during development where possible. During development a small disk image named *nps-2009-canon2-gen2.E01*, located at Digital Corpura[2], was used. This image is approximately 29MB in size and resulted in quick execution of ingest modules. The module was not tested with larger image files, the likes of 1TB or 2TB hard drives, as the size of the image and the subsequent performance of the hashing operation performed on it is outside the scope of this work. Functionally, there should be no difference in the execution of the module on small versus large files. Obviously there will be a performance cost to execute on larger files, as the hashing operation would take longer. The OTS methods, outside of hashing, are relatively constant and should execute within seconds regardless of the data source size.

4.3 Project details

The autopsy-opentimestamps project is open sourced under the GNU LESSER GENERAL PUBLIC LICENSE Version 3, 29 June 2007 and can be found on Github at Weilbach (2017). By making this software open source, it is intended to encourage its use

[1] `https://github.com/opentimestamps/java-opentimestamps/pull/9`

[2] `http://downloads.digitalcorpora.org/corpora/drives/nps-2009-canon2/nps-2009-canon2-gen1.E01`

to validate the proposed use case and to introduce OTS to Autopsy users and the digital forensics community at large. A secondary objective of releasing this module as open source software is to promote further collaboration and refinement on it, and develop it beyond a sample implementation into a fully functional module with improved reporting and user interaction.

At the time of publication, the project had one release in the form of a Netbeans module that can be downloaded and installed in Autopsy version 4.4.0 or later. Table 4.2 lists the other runtime dependencies for the module.

4.4 Summary

In this section the design approach taken to create the autopsy-opentimestamps module was illustrated by looking at the structure of Autopsy modules in general and how that impacts design decisions. The invocation and execution flow of the module was described at a high-level followed by an overview of the project structure. This was accompanied by a discussion of some implementation details, including the difficulties and challenges experienced during development. More detail was provided about the autopsy-opentimestamps project and where it can be located.

Chapter 5

Testing and results

5.1 Introduction

Testing and measurement are of critical importance to determine how effective and efficient solutions are. Based on the research design and the implementation, there are two main testing topics that will be discussed in this section:

- The testing and measurement of the OTS protocol
- The testing and measurement of the functionality in the autopsy-opentimestamps module

The first being which is quantitative in nature. The second being the testing and measurement of the functionality in the autopsy-opentimestamps module which will be quantitative in nature.

5.2 OTS testing results

5.2.1 Data gathering

Data gathering, according to the OTS test design discussed in Section 3.4.1, lasted for 34 days, from 5 September 2017 up to and including 8 October 2017. OTS timestamps were

name	size	fileCreated	fileCreatedTime	formattedTimeUtc	proofCreated	proofCreatedTime	formattedTimeUtc	proofUpgrad	proofUpgradedTime	formattedTimeUtc	proofVerified	proofVerifiedTime	formattedTimeUtc	dataExistedTime	formattedTimeUtc
1504635001.7521157.tst	976.00	CreateFile	1504635001.75	05/09/17 18:10:02	StampFile	1504635003.91	05/09/17 18:10:04	UpgradeFile	1504636205.44	05/09/17 18:30:05	VerifyFile	1504636208.74	05/09/17 18:30:09	1504635891.00	05/09/17 18:24:51
1504635602.0237234.tst	751.00	CreateFile	1504635602.02	05/09/17 18:20:02	StampFile	1504635604.24	05/09/17 18:20:04	UpgradeFile	1504636207.15	05/09/17 18:30:07	VerifyFile	1504636209.16	05/09/17 18:30:09	1504635891.00	05/09/17 18:24:51
1504636201.7237782.tst	994.00	CreateFile	1504636201.73	05/09/17 18:30:02	StampFile	1504636203.74	05/09/17 18:30:04	UpgradeFile	1504637406.18	05/09/17 18:50:06	VerifyFile	1504637408.96	05/09/17 18:50:09	1504637323.00	05/09/17 18:48:43
1504636802.0848937.tst	129.00	CreateFile	1504636802.09	05/09/17 18:40:02	StampFile	1504636804.27	05/09/17 18:40:04	UpgradeFile	1504637407.59	05/09/17 18:50:08	VerifyFile	1504637409.47	05/09/17 18:50:09	1504637323.00	05/09/17 18:48:43
1504637401.908453.tst	197.00	CreateFile	1504637401.91	05/09/17 18:50:02	StampFile	1504637404.37	05/09/17 18:50:04	UpgradeFile	1504639205.01	05/09/17 19:20:05	VerifyFile	1504639209.42	05/09/17 19:20:09	1504637998.00	05/09/17 18:59:58
1504638002.2128801.tst	202.00	CreateFile	1504638002.21	05/09/17 19:00:02	StampFile	1504638004.15	05/09/17 19:00:04	UpgradeFile	1504640405.52	05/09/17 19:40:06	VerifyFile	1504640411.01	05/09/17 19:40:11	1504639087.00	05/09/17 19:18:07
1504638601.987222.tst	223.00	CreateFile	1504638601.99	05/09/17 19:10:02	StampFile	1504638603.65	05/09/17 19:10:04	UpgradeFile	1504640406.64	05/09/17 19:40:07	VerifyFile	1504640411.53	05/09/17 19:40:12	1504639087.00	05/09/17 19:18:07
1504639201.167718.tst	176.00	CreateFile	1504639201.17	05/09/17 19:20:01	StampFile	1504639203.06	05/09/17 19:20:03	UpgradeFile	1504641605.70	05/09/17 20:00:06	VerifyFile	1504641611.27	05/09/17 20:00:11	1504640502.00	05/09/17 19:41:42

Figure 5.1: CSV data loaded into Microsoft Excel for analysis

Table 5.1: Description of data fields extracted from the data set

Field name	**Description**
name	The name of the file which was created
size	The size in bytes of the file that was created
fileCreated	The name of the event
fileCreatedTime	The time (UNIX timestamp) the CreateFile event occurred
proofCreated	The name of the event
proofCreatedTime	The time (UNIX timestamp) the StampFile event occurred
proofUpgraded	The name of the event
proofUpgradedTime	The time (UNIX timestamp) the UpgradeFile event occurred
proofVerified	The name of the event
proofVerifiedTime	The time (UNIX timestamp) the VerifyFile event occurred
dataExistedTime	The time (UNIX timestamp) as of which OpenTimestamps can attest the data existed

created, upgraded, and verified every 10 minutes, resulting in a data set of 4 702 unique files, their timestamps, timestamp results, and operational metadata.

Data Set A

The data set gathered was saved in the complex structure noted in Listing 3.18, and was not easily analysable directly in the database. For this reason, the data was extracted in a flattened format from the database in using a Python script that produced a Comma Separated Values (CSV) output file. The flattened CSV data set has one file per line. A sample can be seen in Figure 5.1 after being imported into Microsoft Excel.

Red columns in Figure 5.1 are unmodified data values, and green columns are calculated fields translating UNIX timestamps into UTC date and time for easily legibility. This data set served as the basis for further analysis, but is also enriched with additional metadata during the analysis stages.

Table 5.1 shows the detailed descriptions of each of the column headers for the base data set.

timeToStamp	timeToUpgrade	timeToVerifyFromStamp	timeToVerifyFromUpgrade	timestampAccuracy
2.15	1201.54	1204.84	3.30	887.09
2.22	602.90	604.91	2.01	286.76
2.02	1202.44	1205.22	2.78	1119.26
2.19	603.32	605.20	1.88	518.73
2.46	1800.63	1805.05	4.41	593.63
1.93	2401.38	2406.87	5.49	1082.85
1.66	1802.99	1807.88	4.89	483.35
1.89	2402.64	2408.21	5.57	1298.94
1.83	1803.62	1807.91	4.29	698.25
1.73	2401.41	2408.32	6.90	988.14
1.88	1805.26	1809.29	4.03	388.75
2.31	1801.88	1806.29	4.41	1099.09
2.11	1203.61	1207.01	3.39	499.41
1.83	1801.40	1806.20	4.79	629.42
2.08	1800.67	1805.61	4.94	1571.91
2.20	1203.04	1206.68	3.65	972.56
1.74	604.72	607.33	2.60	372.72
1.72	1802.12	1806.33	4.22	1085.97

Figure 5.2: Fields calculated from data in Table 5.1

VerifyOperationTime	StampOperationTime
0.231728077	4.49
0.218792677	2.20
0.213683128	1.78
0.237718821	2.46
0.205428362	2.24
0.212933779	2.34
0.25464797	1.95
0.245424032	2.11
0.20843339	2.17
0.218426228	1.74
0.23057127	2.29
0.207798004	6.94
0.237430573	2.31
0.20442009	2.21
0.195673466	2.03
0.200402498	2.06
0.207313299	2.20
0.212749481	1.75
0.197819471	2.27
0.213638306	2.40

Figure 5.3: Fields calculated from data collected in testing logs.

The fields listed in Table 5.1 enabled the calculation of further metrics regarding the performance and accuracy of OTS operations. A sample of these calculated fields can be seen in Figure 5.2.

Table 5.2 gives the detailed descriptions of each of the column headers for the calculated data set. These calculations were performed for each of the 4 703 records.

Data Set B

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 can be seen in Listing A.1 line 240 - 256, and are 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. A sample of this data set can be seen in Figure 5.3.

Table 5.2: Description of data fields calculated from the data in Table 5.1

Field name	Description
timeToStamp	The time, in seconds, it took for a calendar server to confirm the timestamp would be committed, from the time the file was created. This is the time required to create an incomplete OTS timestamp.
timeToUpgrade	The time, in seconds, it took for a complete timestamp to be retrieved, from the time the timestamp was committed on the calendar server (timeToStamp). This is the time required to create a complete OTS timestamp.
timeToVerifyFromStamp	The time, in seconds, it took to verify a complete timestamp, from the time the timestamp was committed on the calendar server (timeToStamp).
timeToVerifyFromUpgrade	The time, in seconds, it took to verify a complete timestamp, from the time the timestamp was committed on the calendar server (timeToVerifyFromStamp).
timestampAccuracy	The time difference in seconds between the time the timestamp was completed (timeToUpgrade) and the time attestation received by the Bitcoin blockchain as per the OTS verify operation.

name	preModVerifyResult	modAction	postModVerfifyResult
1504635001.7521157.tst	TRUE	modFile	FALSE
1504635602.0237234.tst	TRUE	modStamp	FALSE
1504636201.7237782.tst	TRUE	modFile	FALSE
1504636802.0848937.tst	TRUE	modStamp	FALSE
1504637401.908453.tst	TRUE	modFile	FALSE
1504638002.2128801.tst	TRUE	modStamp	FALSE
1504638601.987222.tst	TRUE	modFile	FALSE
1504639201.167718.tst	TRUE	modStamp	FALSE
1504639801.9169676.tst	TRUE	modFile	FALSE
1504640402.1350985.tst	TRUE	modStamp	FALSE
1504641001.3681018.tst	TRUE	modFile	FALSE
1504641601.6086156.tst	TRUE	modStamp	FALSE
1504642201.4844058.tst	TRUE	modFile	FALSE
1504642801.746642.tst	TRUE	modStamp	FALSE

Figure 5.4: A sample of results from the invalidation script execution.

Data Set C

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. A sample of the data set can be seen in Figure 5.4. The modification and validation was performed using another Python script, noted in Listing A.2, which also recorded the results.

The script in Listing A.2 enumerates all of the previously generated files and timestamps, and alternates 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

Table 5.3: Basic timing calculations for Dataset A

	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

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.

5.2.2 Data analysis

Analysis A

Analysis started by looking at the data from Data Set A (Figure 5.2) where various operations and there their timings were recorded. From the perspective of an OTS user, there are a few metrics in this data set that can be significant:

- tStamp: The time it takes to create a timestamp and get a commitment from the calendar server. Includes local processing time.
- tUpgrade: The time it takes to upgrade a timestamp to a complete timestamp. Includes local processing time.
- tVerify: The time it takes to verify a timestamp. Includes local processing time.
- tAccuracy: The accuracy of a timestamp result.

For each of the above-mentioned metrics, an average, minimum, maximum, and standard deviation was calculated and is listed in Table 5.3.

All of the measurement values in Table 5.3 are in seconds and are rounded up to two decimal places. These values will be referred to in coming sections by concatenating the names of the relevant row and column. E.g. The Average (A) timeToUpgrade (tUpgrade) will be denoted by *A-tUpgrade*.

Analysis B

To better analyse the data from Data Set A, the various data points were plotted on a graph. Looking at Figure 5.5, the time to create a complete timestamp has been visualised. 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 5.3.

Similarly, Figure 5.6 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 Figure 5.6 to account for outliers with the overall average *A-tAccuracy* being 2687.64 seconds.

Both of these metrics visualised in Figure 5.5 and Figure 5.6 are relevant to the responsiveness and performance of OTS within the test environment.

Analysis C

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)

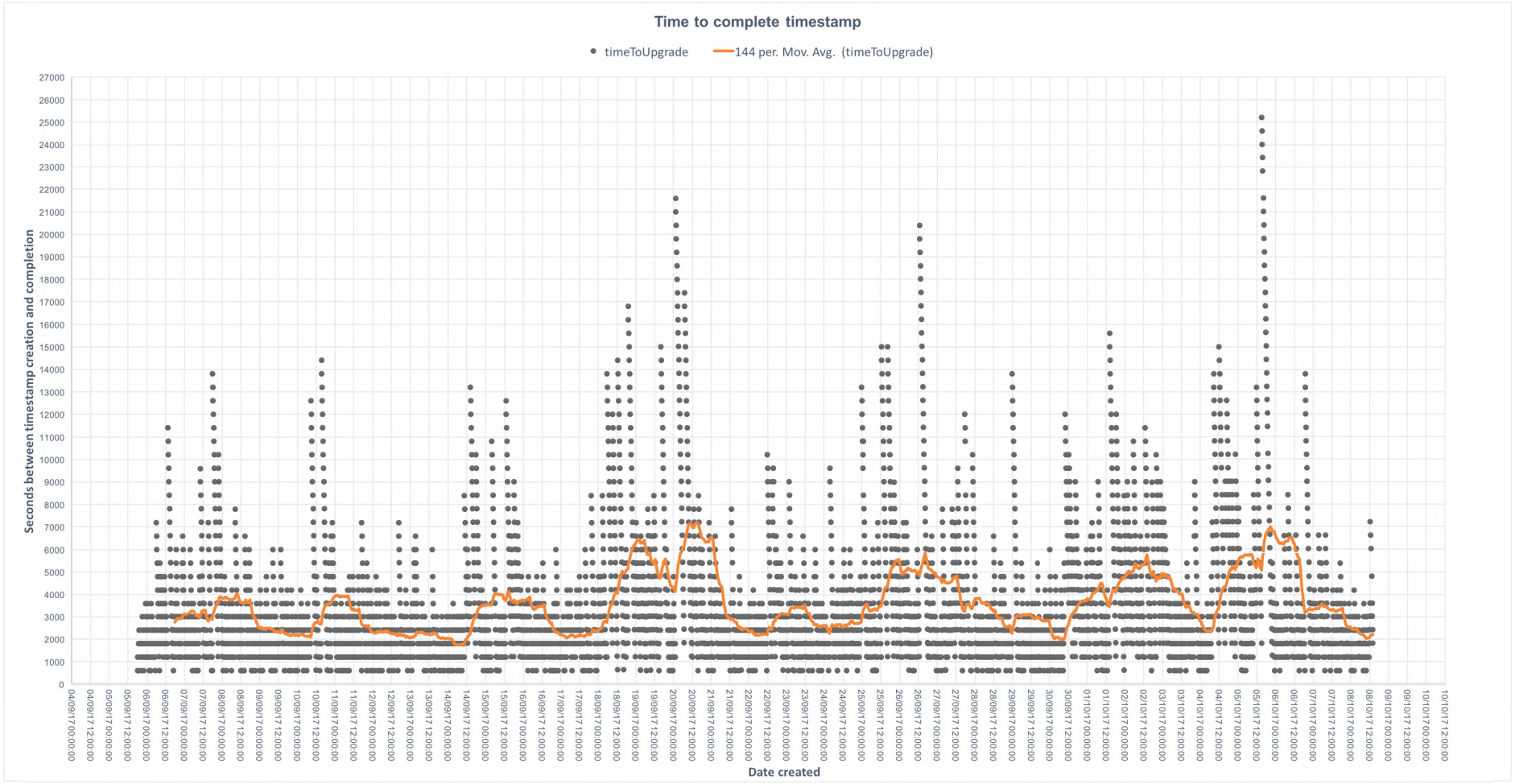


Figure 5.5: Time to complete a timestamp relative to the date and time the timestamp was created.

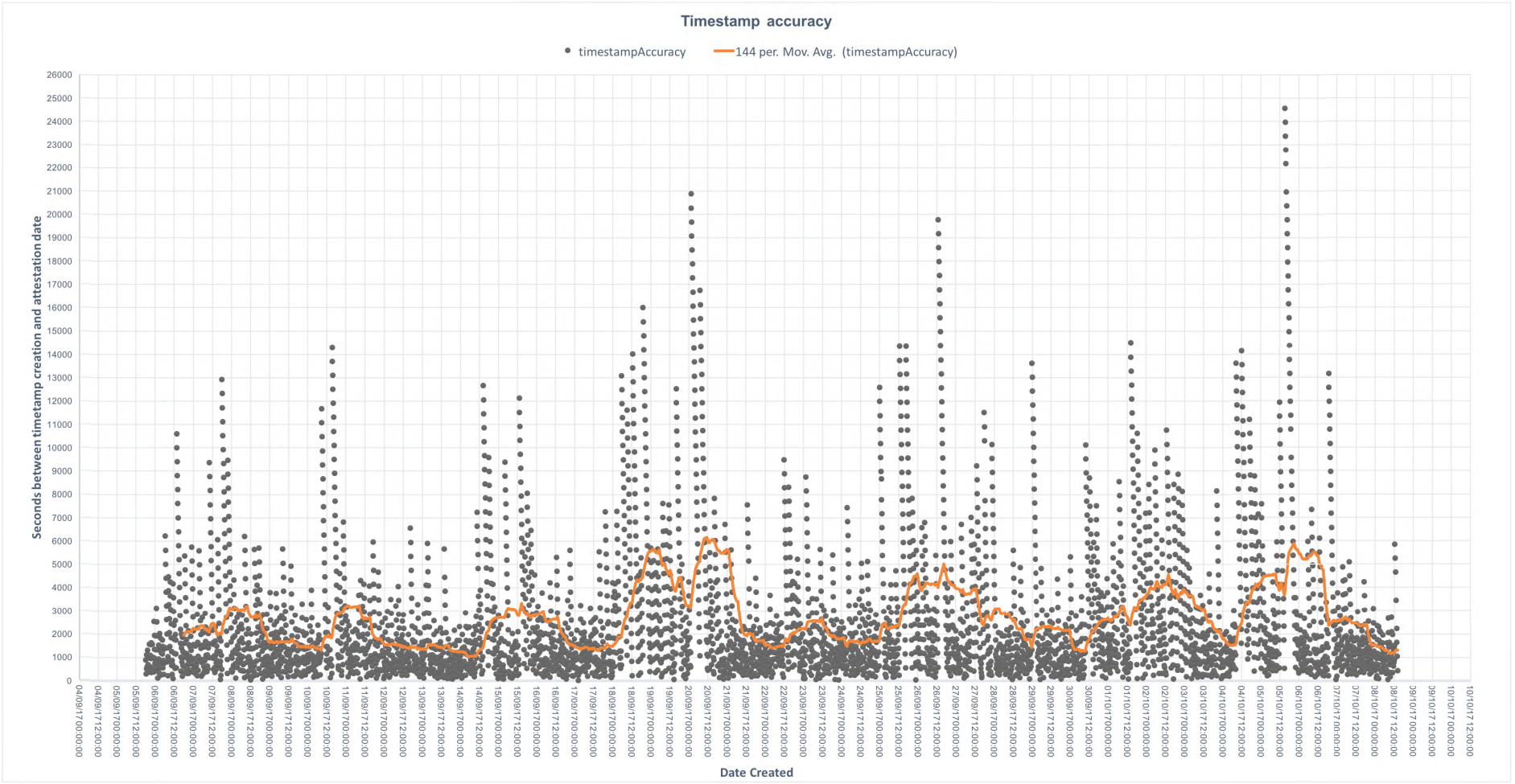


Figure 5.6: Accuracy of a timestamp relative to the date and time the timestamp was created.

Table 5.4: 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

Table 5.5: Error rate of Verify function

No. files tested	Pre-modification result		Modify action	Post-modification result		False pos. result	False neg. result
	True	False		True	False		
2351	2351	0	Modify file	0	2351	0	0
2351	2351	0	Modify timestamp	0	2351	0	0

Table 5.4 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 5.4 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, which amounted to 4 702 data points.

Analysis D

The final piece of analysis was performed on Data Set C, and sought to evaluate the error rate of the OTS Verify function. The Verify function, 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 5.5 shows a summary of the results of this analysis phase.

Column five ('False pos. results') in Table 5.5 shows the number of false positive verification results (zero) whereas column six ('False neg. results'), shows the number of false negative verification results (zero); the meaning of each is discussed in more detail in Section 5.2.3. Both false positive and false negative results are considered significant errors in the Verify function.

5.2.3 Observations and interpretation

In this section, the results of the various analyses performed in Section 5.2.2 are discussed and key observations are drawn from the results.

Performance and timing

Looking at Table 5.3, *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 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 script execution scheduled for 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 Figure 5.7, 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.

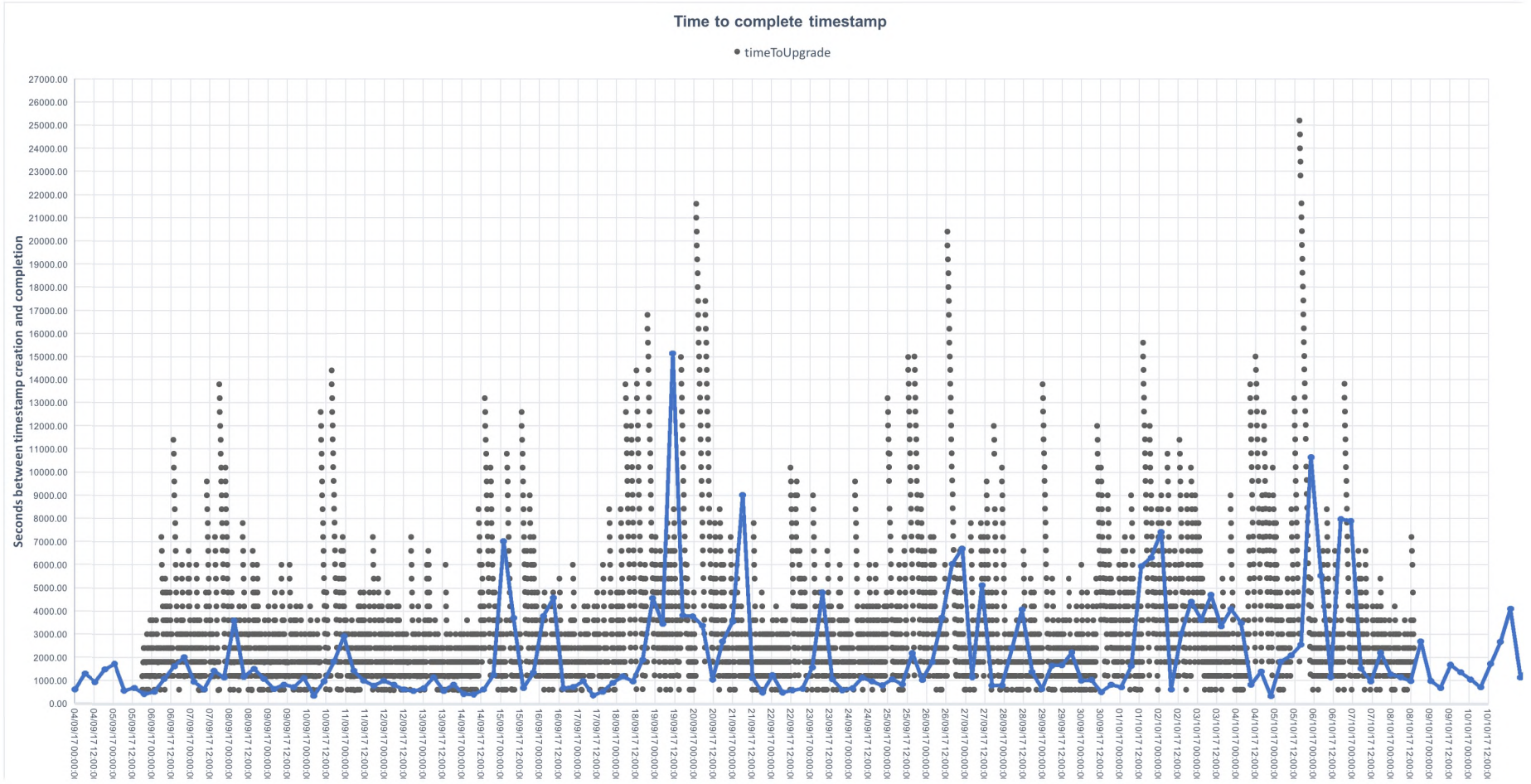


Figure 5.7: Overlay of average block confirmation time from Blockchain Luxembourg S.A (2017a) onto Figure 5.5

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 3.2.4, 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 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 Figure 5.7, 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 Figure 5.6, 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 Figure 5.6, 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 aggregated timestamps and waited 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 Figure 5.6, 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 Figure 5.7, which clearly indicates higher average block confirmation times around these instances of cascading values.

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.

As can seen in Section 5.2.2, 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 from column 5, 'False positive results' in Table 5.5, 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 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. Listing 5.1 shows a small sample of data derived from one such log entry.

Listing 5.1: Populated otsobj

```
1 { "_id" : ObjectId("59da2609b88c3666c45c4f22"), "name" : "1507468801.4121344.tst", "pr
    ↪ oof" : { "info" : "usage: ots info [-h] FILE\nots info: error: argument FILE: c
```

```
↪ an't open '/home/user/ots-tests/testfiles/1507468801.4121344.tst.ots': [Errno
↪ 2] No such file or directory: '/home/user/ots-tests/testfile
↪ s/1507468801.4121344.tst.ots'" } }
```

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 (.tst) 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 Figure 5.8, 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.

5.3 Summary

In this section, the results of the test design discussed in Section 3.4.1 were covered. Some details about how various data sets were gathered and processed were also covered. Three data sets analysed comprised of Data Set A, the main data set with timestamp data; Data

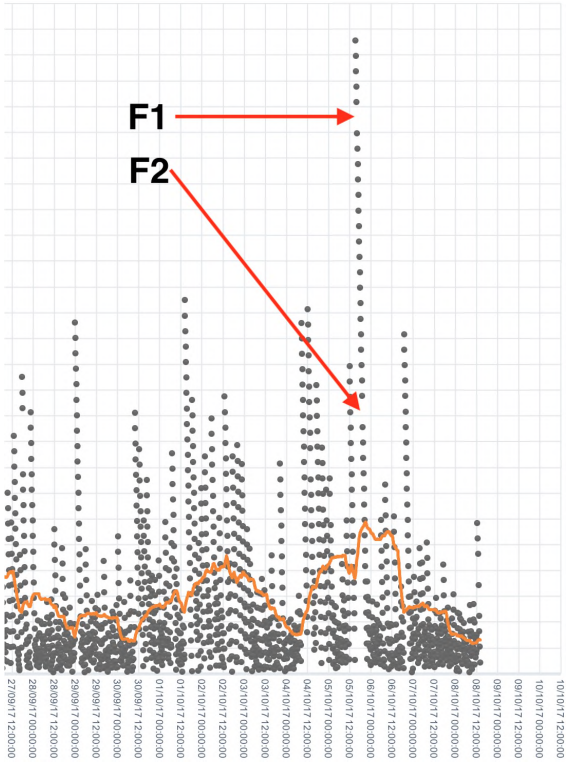


Figure 5.8: Missing timestamps results due to a failure to create timestamp.

Set B, more granular measurements of OTS operations; and Data Set C, the error rate of OTS.

Analysis was performed on each of the data sets in turn, and basic values, such as average timestamp creation time, based on Data Set A in Analysis A, were calculated. Analysis B, was performed by plotting Data Set A values on a graph to better visualise the distribution of the values. It was observed that the time to create a complete timestamp, and the timestamp accuracy were heavily influenced by the block confirmation times. Moving averages for both cases were calculated and it was observed that, despite a few outlier values, the average time for both operations were fairly consistent in aggregate.

In Analysis C, more accurate metrics around specific OTS operation were calculated, and it was concluded that these were consistent as they lack any interaction with the Bitcoin blockchain (which would introduce variability). It was also noted that script overhead time was small enough to be insignificant.

During Analysis D, data relating to the false positive and false negative error rate of the OTS verification operation was presented.

In the paragraph on performance and timing it was reiterated that OTS performance

was, for the most part, consistent in the test sample, and that the biggest influence on the operation times is the performance of the Bitcoin network. It was shown that a correlation between the test result times and the network performance can be drawn by overlaying average block confirmation data over the data set. Importantly, the cascading effect was observed and explained as being caused by inconsistent and slow block confirmation times, as opposed to fixed timestamp creation times in the test script.

This section was concluded by discussing the error rate observed for OTS, and how there are two error rate measurements. The error rate for verification which is very important, and the error rate for creation which is less important. It was shown that the error rate for the OTS Verify function is virtually 0 and was insignificant in the test sample. It was also noted that the error rate for creation was not specifically measured, but was observed to approximately 0.55% out of the test sample.

Chapter 6

Conclusions and recommendations

6.1 Research objectives

Review of candidate technologies

The first research objective noted in Section 1.2, was to review candidate technologies. This specific objective was to identify existing candidate technologies from the common body of knowledge. It further aimed to review and compare a subset of these technologies to serve as foundation for the research going forward. Finally, it would aim to identify a specific technology to use as part of the remaining objectives. This objective was supported by a background discussion on digital forensics (Section 2.1), and blockchain technology (Section 2.2), the two main themes of this work. By providing the background, the reader was equipped with the necessary knowledge to understand the problems pertaining to evidence integrity in digital forensic practices, as well as the beneficial properties and drawbacks of blockchain technology. This background knowledge is essential to critically evaluate further research on the topic and to evaluate potential candidate technologies.

With the background established, existing literature combining distributed ledger technology and evidence integrity aspects of digital forensics was discussed. It was noted that the practice of embedding data in the Bitcoin blockchain as an immutable store is not novel and has been done for almost as long as Bitcoin has existed. Work by Okupski (2015), that proposed a more efficient method for embedding data on the Bitcoin blockchain,

was shown. This led to a discussion on research specifically aimed at embedding digital evidence in the Bitcoin blockchain, albeit in a narrow scope.

Finally, in Section 2.3.6, a sample of protocols and services that was specifically aimed at providing notary, or timestamping services were identified and compared. A subset of these were compared on the basis of transparency, compatibility and accessibility, and it was decided that OTS - being an open source protocol, service and implementation - would be the ideal candidate technology to base the research effort on. This selection was followed by a brief overview of the OTS project, some use cases, and the purpose of the project, all of which was entirely supportive of the stated research effort.

The objective was concluded with an in depth technical analysis of the OTS protocol, and OTS implementation in Section 3.2. This detailed analysis gave the user a low-level understanding of how OTS functions and, by extension, justifying the trust they can potentially have in OTS. A number of possible OTS configuration options were presented and a discussion held on how each impacted the usability vs. security equation. A discourse on how OTS addresses issue of timing, accuracy, privacy, and security followed in Section 3.2.6. Further exploration of a highly usable, but less trustworthy configuration involving public calendar servers as the basis for further work was proposed, given the need for easy adoption of the technology. It was concluded that OTS is a suitable solution, that is both transparent and provably trustworthy given a certain configuration.

Proof-of-Concept implementation of the chosen technologies

This research objective was to implement OTS into a digital forensic tool. It included a short section on the selection criteria for the tool of choice (Section 3.3). This included a discussion on why and how this choice was appropriate. The modular structure and open nature of Autopsy was covered in Section 3.3. It was concluded that Autopsy is used and supported by an active community, and is therefore an ideal conduit for potential adoption of OTS. A basic solution design was outlined which comprised a minimal use case diagram (Figure 3.9), showing how potential actors would create and verify OTS timestamps.

Once a suitable tool was chosen and discussed, the specific development environment configured for implementation to support possible future efforts was covered. This section concluded with a discussion on the entire IDE and dependencies applicable for the implementation effort. This was followed by an overview of Autopsy module development in

general, to get a better idea of the implementation options available. It was found that there are many module development models, as well as resources for supporting module development available to developers. Java was selected as the implementation language as a result of OTS dependencies and Autopsy's existing reliance on Java.

A more specific module design was discussed, and it found the most appropriate module for the proposed use case would be a data source ingest module, due to the nature and timing of its invocation and data sources as imported into Autopsy. It was discovered that different data sources had to be processed differently, as a result of their structure, and an appropriate solution was implemented to allow the timestamping of non-hierarchical files structures. A process flowchart of the module execution was developed and can be seen in Figure 4.3. Logging and reporting specific to the autopsy-opentimestamps module was also highlighted in this section. Finally, some implementation details and challenges experienced during implementation were discussed.

We concluded by publicly releasing an initial alpha build of the software, and noting where it can be found for use and further collaborative development. The research objective was met as expected, by developing and releasing to the public an implementation of OTS in a popular digital forensic tool. This proved that with the modular nature of Autopsy combined with the succinct power and simplicity of OTS, that creating a trustworthy immutable and easily verifiable timestamp of digital artefacts was practical.

Measurement and testing of the chosen technology and implementation

Before implementation was attempted for autopsy-opentimestamps, a robust and exhaustive test design was undertaken to ensure that there was a capability to measure the performance of OTS at scale and in specific instances.

The test design, proposed in Section 3.4.1, outlined an approach for creating and verifying OTS timestamps at scale through automation. This automation ensured that a large data set could be gathered for further analysis. The test design proposed and detailed a very specific and dedicated test environment for the execution of the tests, and noted the configuration in Table 3.2.

Specifics about the test script were noted, and the stored data structure was detailed to outline which values were captured and how they were collected. Some attention was also drawn to the execution flow of the test and some of the criteria used to determine the state of a timestamp during the test execution. Redundant data was captured as a

precautionary measure, and extensive logging was performed as part of the testing effort. The test design was concluded by covering the timing of the test script execution, and why that timing was significant to prevent concurrent execution and data possible corruption.

Following the test design, the test data gathering and analysis was performed. Specific valuable data fields were extracted from the data set that consisted of 4 702 unique timestamps created over a period spanning more than a month. Multiple smaller, more specific data sets were derived from the base data set to form a clearer picture of OTS performance. Data Set A, Data Set B, and Data Set C were extracted, and details about the values in these data sets were discussed. Subsequent to the data set clean-up, various analyses were performed on the data sets in question. Analysis A aimed to revealed some basic statistics about key performance indicator of OTS. Analysis B provided a visualisation of a few significant key performance indicators by plotting data point on a graph. Analysis C revealed more granular OTS performance metrics and Analysis D, significantly, explored the error rates in OTS execution.

Testing revealed that OTS is a stable and reliable protocol and implementation with relatively consistent performance primarily dependent on the performance of the Bitcoin network. Execution times for operations such as creation and verification, excluding local processing time, were very fast and consistent. Interaction with the public calendar server did affect the consistency of performance negatively and illustrated perfectly the usability versus security trade-off.

Errors were classified as either creation errors or verification errors. The former being more serious than the latter, as a consequence of the perceived trust in the OTS system. It was revealed that creation errors for the measured sample occurred 0.553% of the time, but that the impact of such errors is limited and can easily be mitigated by performing more extensive validation of timestamp creation before accepting that a timestamp has been created. The root cause of the creation errors could not be definitively established due to a lack of detail in the logs. Verification error rates were insignificant at 0% in the data set, reinforcing the belief that the OTS verification mechanism is robust and the timestamp fragile enough to result in virtually error free system.

6.2 Limitations

Accuracy

As discussed in Section 3.2.6, there is an inherent inaccuracy in the Bitcoin block header confirmation time: "We can therefore conclude that although the timestamp attests with accuracy up to the second, the timestamp should be interpreted as accurate within two to three hours and accurate to within a day in a worst case scenario." (Todd, 2017). Combining this with the calendar server aggregation and timestamping lag as discussed in Section 5.2.2, the accuracy of timestamps up to the minute or hour is misleading and OTS timestamps should be considered accurate up to the day. This inaccuracy of OTS may result in it being unusable for use cases where more accurate timestamps are necessary. In cases where the timeline of events is short, I.E., when the time a timestamped email received is relevant up to the second, OTS will not be valuable. If, however, the fact that a timestamped email was sent on a day prior to another event before the commencement of an investigation, OTS would be very useful. The issue of accuracy is not so much a challenge as it is a limitation that potential users should be aware of.

Cost

As the price of Bitcoin continues to fluctuate, with significant and ongoing price increases at the time of writing, one has to consider the future feasibility of OTS as it carries a Bitcoin cost as discussed in Section 2.3. During the time this research was conducted, the price per Bitcoin increased from $1 097.98 to $7 392.97. If the price of a Bitcoin continues to inflate at the current rate, the cost of a non-spendable Bitcoin transaction will be prohibitively high, even when employing data aggregation at the cost of accuracy. The success of Bitcoin and subsequent significant price increases may result in future cost-benefit analyses, indicating that the cost of embedding data in the Bitcoin blockchain is simply too high. Once the smallest denomination of Bitcoin, that can be transacted, has a higher cost than the value derived from embedding data, OTS will no longer be feasible. At the very least, a continued increase in the Bitcoin price could result in the closure of some public OTS calendar servers, as the operator of such a calendar server relies on their own funds and donations to support OTS operations and the associated Bitcoin transactions.

6.3 Future work

This research aims to serve as a base for further research and development of OTS in a digital forensic context, and thus leaves ample opportunity for future work. Future work can be divided into two categories:

- Future work on OTS in general, and
- Future work on the autopsy-opentimestamps OTS implementation

autopsy-opentimestamps module

As Todd (2016b) noted, OTS is in public alpha release and is actively being developed and maintained, and can thus be expected to be expanded upon in the near future. The autopsy-opentimestamps module developed as part of this work implemented only base functionality of OTS, ignoring some more specialised configurations such as proxy awareness, whitelisting, and custom Bitcoin node configuration. Autopsy-opentimestamps can be further improved on to support all of these configuration options from the module settings interface. Furthermore, autopsy-opentimestamps can be made to support future OTS features as and when they are released.

As identified in Section 4.1.3, the reporting functionality as it exists in autopsy-opentimestamps upon its public release, is text-based and fairly basic. This basic reporting can be vastly improved on by better integrating the OTS reporting into Autopsy using more complex and visually appealing presentation mechanisms. The reporting format can also be structured better to allow for easier parsing by other systems.

Finally, autopsy-opentimestamps can also be enhanced to support easy export and the sharing of data source timestamps. One of the main purposes of autopsy-opentimestamps was to make it easy for multiple parties to verify data source integrity against a trusted source, and having the ability to easily and securely export timestamps for distribution among other stakeholders would support this objective. Timestamps themselves do not contain any sensitive data about the data source other than the hash, and can therefore readily be shared with other parties already in possession of the source data.

OTS in general

The success of OTS as an aid to digital forensic processes depends heavily on its adoption and acceptance as adoption by the community of researchers and practitioners is the first step to further acceptance and trust in legal realms and courtrooms. OTS, being open source and supported on many platforms, is the ideal candidate for implementation in a wider range of open source and commercial digital forensics tools.

Future work can include integrations with other popular digital forensics tools such as the Volatility framework, or commercial tools such as FTK Imager. OTS can also be integrated at a protocol level into utilities such as AFFlib (Sshock, 2017), which creates and stores data in the Advanced Forensic Format (AFF). By supporting OTS timestamps at a file format level in AFF, the utility of OTS will potentially be appreciated by a wider audience and serve as the de facto timestamping format for digital evidence.

OTS was designed as an open and extensible protocol that could be used with other blockchains. The implementation chosen by the OTS author, Todd (2016b), was built on the Bitcoin blockchain since its immutability and stability were trusted over that of other blockchains, the Ethereum blockchain for instance. This, however, does not mean that OTS works exclusively on the Bitcoin blockchain. The OTS architecture and design is blockchain-agnostic with the validation structure abstracted from the blockchain interaction. This means that the same timestamping mechanism can be used with any other blockchain with some modification. The eventual accuracy and trust of such a timestamp would then change and depend on the immutability and trust of the supporting blockchain. Future work could include implementing OTS on other blockchains or creating OTS timestamps redundantly on many blockchains for multiple sources of trust.

6.4 Summary

During this research, OTS was implemented in a quick and reliable manner to show how it can possibly be leveraged in other tools. The open nature of OTS is not only beneficial to establishing trust, but also to easy integration with open source and commercial tools. We showed that the act of timestamping is no longer novel, and that while many such services exists in the public and commercial realm, OTS, with its open nature and robust design, is a very good candidate for use in systems that require transparency and trust. By using OTS, it would be possible to easily create, verify, and share timestamps of

which the integrity can be proven and which can be accessed universally. By enabling and illustrating this level of trust and transparency in the context of digital evidence integrity verification, it will catalyse wider adoption of the technology. By using such as system, namely one based on OTS, parties to an investigation can independently create but collaboratively verify the integrity of digital artefacts in a way that is sufficiently accurate and provably trustworthy.

Bibliography

Anz and IBM. Distributed ledger technology and Bank guarantees for commercial property leasing. July 2017.
URL https://www.westpac.com.au/content/dam/public/wbc/documents/pdf/aw/media/WhitePaper_Blockchain_PoC_for_Bank_Guarantees.pdf

Araoz, M. and Ordano, E. Proof of Existence. 2013.
URL http://proofofexistence.com/

Arizona State Legislature, Fifty-third Legislature, First Regular Session. House bill 2417. 2017.
https://apps.azleg.gov/BillStatus/GetDocumentPdf/452616.

Back, A. Hashcash - A Denial of Service Counter-Measure. 2002.
URL http://www.hashcash.org/papers/hashcash.pdf

Barrdear, J. and Kumhof, M. The Macroeconomics of Central Bank Issued Digital Currencies. *SSRN Electronic Journal*, (605), 2016. ISSN 15565068. doi:10.2139/ssrn.2811208.

Beel, J., Breitinger, C., Langer, S., Lommatzsch, A., and Gipp, B. Towards reproducibility in recommender-systems research. *User Modeling and User-Adapted Interaction*, 26(1):69–101, 2016. ISSN 15731391. doi:10.1007/s11257-016-9174-x.

Berghel, H. The discipline of Internet forensics. *Communications of the ACM*, 46(8):15, 2003. ISSN 00010782. doi:10.1145/859670.859687.

Bitcoin Foundation. Wallet: Pruning. 2016.
URL https://github.com/bitcoin/bitcoin/blob/v0.12.0/doc/release-notes.md#wallet-pruning

Bitcoin Project. Running A Full Node. 2015.
URL https://bitcoin.org/en/full-node

Bitcoinwiki. Controlled supply. 2014.
URL https://en.bitcoin.it/wiki/Controlled_supply

Blockchain Luxembourg S.A. Average block confirmation time. 2017a.
URL https://blockchain.info/charts/avg-confirmation-time?timespan=60days&showDataPoints=true

Blockchain Luxembourg S.A. Blockchain. 2017b.
URL https://blockchain.info/

Brilliant.org. Merkle Tree. 2015.
URL https://brilliant.org/wiki/merkle-tree/

Campbell, R. Arizona's Blockchain Bill Passes Senate Leading the Way to Smart Contracts. 2017.
URL https://www.cryptocoinsnews.com/arizonas-blockchain-bill-passes-senate-leading-way-smart-contracts/

Carrier, B. Open Source Digital Forensics Tools : The Legal Argument. Technical Report October, 2002.
URL http://home.eng.iastate.edu/~guan/course/backup/CprE-592-YG-Fall-2002/paper/atstake_opensource_forensics.pdf

Crespo, A. S. d. P. and García, L. I. C. Stampery blockchain timestamping architecture (bta)-version 6. *arXiv preprint arXiv:1711.04709*, 2017.

Curry, D. DHS looking to link to the blockchain. 2016.
URL http://readwrite.com/2016/07/15/homeland-security-blockchain-research-pl1/

del Castillo, M. The DAO: Or How A Leaderless Ethereum Project Raised $50 Million. 2016.
URL http://www.coindesk.com/the-dao-just-raised-50-million-but-what-is-it/

Delmolino, K., Arnett, M., Kosba, A., Miller, A., and Shi, E. Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab. In *International Conference on Financial Cryptography and Data Security*, pages 79–94. Springer, 2016.

Dykstra, J. and Sherman, A. T. Acquiring forensic evidence from infrastructure-as-a-service cloud computing: Exploring and evaluating tools, trust, and techniques. *Digital Investigation*, 9:90–98, 2012. ISSN 17422876. doi:10.1016/j.diin.2012.05.001.

Dykstra, J. and Sherman, A. T. Design and implementation of frost: Digital forensic tools for the openstack cloud computing platform. *Digital Investigation*, 10:S87–S95, aug 2013. ISSN 17422876. doi:10.1016/j.diin.2013.06.010.

Ethereum Foundation. Ethereum. 2016.
URL `https://www.ethereum.org/`

Everledger Ltd. Welcome to the digital vault of the future. 2017.
URL `https://www.everledger.io/`

Filippi, P. D. Bitcoin: a regulatory nightmare to a libertarian dream. *Internet Policy Review*, 3(2):43, 2013. doi:10.14763/2014.2.286.

Finley, K. A $50 Million Hack Just Showed That the DAO Was All Too Human. 2016.
URL `https://www.wired.com/2016/06/50-million-hack-just-showed-dao-human/`

Garfinkel, S. L. Digital forensics research: The next 10 years. *Digital Investigation*, 7, 2010. ISSN 17422876. doi:10.1016/j.diin.2010.05.009.

Gipp, B., Meuschke, N., and Gernandt, A. Decentralized Trusted Timestamping using the Crypto Currency Bitcoin. *iConference 2015*, pages 1–6, 2015.

Gottehrer, G. "Connected" Discovery: What the ubiquity of digital evidence means for lawyers and litigation. *Richmond Journal of Law & Technology*, XXII(3):1–27, 2016. ISSN 02729490. doi:10.1525/sp.2007.54.1.23.

Harper, J. Understanding Bitcoin's Scaling Debate: Politics Comes First. 2017.
URL `https://www.coindesk.com/understanding-bitcoins-scaling-debate-politics-comes-first/`

InfoSec Institute. 22 Popular Computer Forensics Tools. 2017.
URL `http://resources.infosecinstitute.com/computer-forensics-tools/`

Käsper, E. Fast elliptic curve cryptography in openssl. In *Proceedings of the 2011 International Conference on Financial Cryptography and Data Security*, FC'11, pages 27–39. Springer-Verlag, Berlin, Heidelberg, 2012. ISBN 9783642298882. doi:10.1007/978-3-642-29889-9_4.

Kessler, G. C. Anti-Forensics and the Digital Investigator. *Proceedings of the 2014 47th Hawaii International Conference on System Sciences*, pages 1–7, 2006. doi:10.4225/75/57ad39ee7ff25.

Kiayias, A. and Panagiotakos, G. Speed-Security Tradeoffs in Blockchain Protocols. *School of Informatics University of Edinburgh*, page 19, 2015.

Kuntze, N., Rudolph, C., Alva, A., Endicott-Popovsky, B., Christiansen, J., and Kemmerich, T. On the creation of reliable digital evidence. In *IFIP Advances in Information and Communication Technology*, volume 383 AICT, pages 3–17. 2012. ISBN 9783642339615. ISSN 18684238. doi:10.1007/978-3-642-33962-2_1.

Lemieux, V. L. Blockchain and distributed ledgers as trusted recordkeeping systems. *Future Technologies Conference (FTC) 2017*, 2017.

Litecoin Project. What Is Litecoin? 2017.
URL `https://litecoin.org/`

Malkovský, M. The Concept of Smart Contracts. Technical Report April, University of Turku, 2015.

Manson, D., Carlin, A., Ramos, S., Gyger, A., Kaufman, M., and Treichelt, J. Is the open way a better way? Digital forensics using open source tools. *Proceedings of the Annual Hawaii International Conference on System Sciences*, pages 1–10, 2007. ISSN 15301605. doi:10.1109/HICSS.2007.301.

Marc, P. Blockchain Technology: Principles and Applications. In *Handbook of Research on Digital Transformations' edited by F. Xavier Olleros, and Majlinda Zhegu.* Edward Elgar, 2016. Handbook of Research on Digital Transformations edited by F. Xavier Olleros, and Majlinda ZheguA paraitre.
URL `https://halshs.archives-ouvertes.fr/halshs-01231205`

Merkle, R. C. Protocols for Public Key Cryptography. *Synopsis on Security and Privacy*, pages 122–134, 1980.

MongoDB Inc. Install MongoDB Community Edition on Ubuntu. 2016.
URL `https://docs.mongodb.com/manual/tutorial/install-mongodb-on-ubuntu/`

Nakamoto, S. Bitcoin: A Peer-to-Peer Electronic Cash System. 2008.
URL `https://bitcoin.org/bitcoin.pdf`

Okupski, K. (Ab) using Bitcoin for an Anti-Censorship Tool. Ph.D. thesis, Eindhoven University of Technology, 2015.

Opentimestamps. OpenTimestamps Client. 2017.
URL `https://github.com/opentimestamps/opentimestamps-client/blob/master/README.md`

Opentimestamps.org. A timestamping proof standard. 2017.
URL `https://opentimestamps.org/`

Peters, G. W. and Panayi, E. Understanding modern banking ledgers through blockchain technologies: Future of transaction processing and smart contracts on the internet of money. *New Economic Windows*, pages 239–278, 2016. ISSN 20394128. doi:10.1007/978-3-319-42448-4_13.

Preneel, B. Cryptographic hash functions. *European Transactions on Telecommunications*, 5(4):431–448, 1994. ISSN 15418251. doi:10.1002/ett.4460050406.

Rampton, J. 101 Top Blockchain Companies. 2016.
URL `https://due.com/blog/101-top-blockchain-companies/`

Redman, J. Popular Bitcoin Exchanges Reveal Controversial Hard Fork Contingency Plan. 2017.
URL `https://news.bitcoin.com/popular-bitcoin-exchanges-reveal-controversial-hard-fork-contingency-plan/`

Rogaway, P. Nonce-Based Symmetric Encryption, pages 348–358. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004. ISBN 978-3-540-25937-4. doi:10.1007/978-3-540-25937-4_22.

Sabernick III, B. A. Development of an Autopsy Forensics Module for Cortana Artifacts Analysis. *International Journal of Computer Science and Information Security*, 14(7):111–121, 2016. ISSN 19475500.

Schneier, B. Applied Cryptography: Protocols, Algorithms, and Source Code in C. John Wiley & Sons, Inc., New York, NY, USA, 1993. ISBN 0471597562.

Sleuthkit.org. Autopsy Developer's Guide. 2015.
URL `https://wiki.sleuthkit.org/index.php?title=Autopsy_Developer%27s_Guide`

Sleuthkit.org. Autopsy 3rd Party Modules. 2016.
URL https://wiki.sleuthkit.org/index.php?title=Autopsy_3rd_Party_Modules

Sleuthkit.org. Autopsy. 2017a.
URL http://sleuthkit.org/autopsy/docs/api-docs/4.4/

Sleuthkit.org. Autopsy User Documentation. 2017b.
URL http://sleuthkit.org/autopsy/docs/user-docs/4.5.0/

Sshock. AFFLIBv3. 2017.
URL https://github.com/sshock/AFFLIBv3

Tobergte, D. R. and Curtis, S. Guide to Computer Forensics and Investigations, volume 53. Cengage Learning, 2013. ISBN 9788578110796, 1689–1699 pages. doi: 10.1017/CBO9781107415324.004.

Todd, P. [bitcoin-dev] Interpreting nTime for the purpose of Bitcoin-attested timestamps. 2016a.
URL https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2016-September/013120.html

Todd, P. OpenTimestamps: Scalable, Trustless, Distributed Timestamping with Bitcoin. 2016b.
URL https://petertodd.org/2016/opentimestamps-announcement

Todd, P. Opentimestamps masters research project. Personal communication, 2017.

Valjarevic, A. and Venter, H. S. Towards a Digital Forensic Readiness Framework for Public Key Infrastructure systems. *2011 Information Security for South Africa - Proceedings of the ISSA 2011 Conference*, pages 1–10, aug 2011. doi:10.1109/ISSA.2011.6027536.

Valjarevic, A. and Venter, H. S. Implementation guidelines for a harmonised digital forensic investigation readiness process model. *2013 Information Security for South Africa - Proceedings of the ISSA 2013 Conference*, pages 1–9, aug 2013. doi:10.1109/ISSA.2013.6641041.

Volf, M. What is the difference between Litecoin and Bitcoin? 2016.
URL http://www.coindesk.com/information/comparing-litecoin-bitcoin/

Wayne, V., Wilkinson, S., and Bukowski, J. Chainpoint: A scalable protocol for recording data in the blockchain and generating blockchain receipts, 2016.
URL `https://tierion.com/chainpoint`

Weilbach, T. Bots , Block Chains and Believable Logs. Technical report, University of South Africa, Pretoria, 2014.

Weilbach, T. autopsy-opentimestamps. 2017.
URL `https://github.com/tweilbach/autopsy-opentimestamps`

Wijaya, D. A. and Suwarsono, D. A. Securing Digital Evidence Information in Bitcoin. Technical report, Monash University Melbourne, Australia, 2016.

Wilson, C. Digital Evidence Discrepancies: Casey Anthony Trial. 2011.
URL `http://www.digital-detective.net/digital-evidence-discrepancies-casey-anthony-trial/`

Witte, J. H. The Blockchain: A Gentle Introduction. pages 1–5, 2016. ISSN 15565068. doi:10.2139/ssrn.2887567.

Wood, G. Ethereum: a secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper*, pages 1–32, 2014. ISSN 10986596. doi:10.1017/CBO9781107415324.004.

Zerocoin Electric Coin Company. About Us. 2016.
URL `https://z.cash/about.html`

Appendix A

Code listings

A.1 ots-test.py

Listing A.1: ots-test.py source code

```
1  #!/usr/bin/env python3
2
3  import os
4  from random import randint
5  import logging
6  from pymongo import MongoClient
7  import subprocess
8  import time
9  from time import mktime
10 from datetime import datetime
11
12 logger = logging.getLogger(__name__)
13 logger.setLevel(logging.INFO)
14
15 # create a file handler
16 handler = logging.FileHandler('ots-test.log')
17 handler.setLevel(logging.INFO)
18
19 # create a logging format
20 formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s')
21 handler.setFormatter(formatter)
22
23 # add the handlers to the logger
24 logger.addHandler(handler)
```

```
25
26 # setup connection to mongo instance
27 client = MongoClient("mongodb://[user]:[password]@localhost/otsdata")
28 db = client.otsdata
29
30
31 testFilePath = os.path.join(os.getcwd(), 'testfiles')
32
33 def insertOtsObj(file):
34     res = db.otsfiles.insert_one(file)
35     print(res)
36
37
38 def getRandomInt ():
39     return randint(1, 1024)
40
41
42 def writeRandomFile (path):
43     filePath = os.path.join(path, str(time.time()) + '.tst')
44     with open((filePath), 'wb') as fout:
45         fout.write(os.urandom(getRandomInt()))
46     return filePath
47
48
49 def getProofTimestamp(proofstring):
50     # split the string and get the time component
51     tstime = proofstring.split("as of ", 1)[1]
52     # format the time string
53     time_tuple = time.strptime(tstime, '%a %b %d %H:%M:%S %Y %Z')
54     # Make a datetime object from fortmatted time
55     dt = datetime.fromtimestamp(mktime(time_tuple))
56     #return timestamp
57     return dt.timestamp()
58
59
60 def getLocalTimeFromEpoch (epoch):
61     return time.strftime('%Y-%m-%d %H:%M:%S:%f', epoch)
62
63 def stamp (file):
64     try:
65         start = time.time()
66         res = subprocess.getoutput('/usr/local/bin/ots stamp ' + file)
67         stop = time.time()
68         logger.info("Stamp time: %s", str(stop-start))
```

```
69          print(res)
70          return True
71      except:
72          return False
73
74  def info (file):
75      res = ''
76      try:
77          res = subprocess.getoutput('/usr/local/bin/ots -v info ' + file)
78          print(res)
79          return res
80      except:
81          return res
82
83  def upgrade (file):
84      start = time.time()
85      res = subprocess.getoutput('/usr/local/bin/ots upgrade ' + file)
86      stop = time.time()
87      logger.info("Upgrade time: %s", str(stop-start))
88      print (res)
89      if res.__contains__("Success! Timestamp complete"):
90          return True
91      else:
92          return False
93
94  def verify(file):
95      start = time.time()
96      res = subprocess.getoutput('/usr/local/bin/ots verify ' + file)
97      stop = time.time()
98      logger.info("Verify time: %s", str(stop-start))
99      print(res)
100     return res
101
102 def getProofPath (file):
103     return file + '.ots'
104
105 def parseVerifiedDate(date):
106     parts = date.split()
107
108 #print(json.dumps(Encoder().encode(otsdata)))
109 def create_new_file():
110
111     otsevent = {
112         'name': '',
```

```
113         'time': '',
114         'command': '',
115         'message': ''
116     }
117     otsproof = {
118         'created': False,
119         'committed': False,
120         'info': '',
121         'upgraded': False,
122         'verified': False
123     }
124     otsobj = {
125         'name': '',
126         'path': '',
127         'proof': '',
128         'events': [],
129         'size': ''
130     }
131     filePath = writeRandomFile(testFilePath)
132     otsobj['events'].append({
133         'name': 'CreateFile',
134         'time': time.time(),
135         'command': 'create',
136         'message': 'Created new file ' + filePath
137     })
138
139     #stamp that thaang
140     committed = stamp(filePath)
141     #committed = False
142
143     #update otsproof
144     otsproof['created'] = True
145     otsproof['createdTime'] = time.time()
146     otsproof['committed'] = committed
147     otsproof['info'] = info(getProofPath(filePath))
148
149     otsobj['events'].append({
150         'name': 'StampFile',
151         'time': time.time(),
152         'command': 'stamp',
153         'message': 'Stamped file '+filePath+' with result '+ str(committed)
154     })
155     # proofpath = getProofPath(filePath)
156     # print(proofpath)
```

```
157     logger.info("Stamp created for %s", filePath)
158     otsobj['name'] = os.path.basename(filePath)
159     otsobj['path'] = filePath
160     otsobj['proof'] = otsproof
161     otsobj['size'] = os.path.getsize(filePath)
162     # formulate otsfile object
163     # Add it to the db
164     insertOtsObj(otsobj)
165 
166 def verify_timestamp():
167     otsobjs = db.otsfiles.find({"proof.upgraded": True, "proof.verified": False})
168 
169     # loop through all the non-upgraded stamps and upgrade them
170     for otsobj in otsobjs:
171         # print(str(otsobj))
172         # get the object id so we can reference it later
173         _id = otsobj['_id']
174         #print(_id)
175         #print(otsobj)
176         #print(getProofPath(otsobj['path']))
177         logger.info("Found obj ready to verify at %s with ID %s" % (getProofPath(otsob
                ↪ j['path']), _id))
178 
179         res = verify(getProofPath(otsobj['path']))
180         logger.info("Found obj with ID %s. Verification result %s" % (_id, res))
181 
182         if res.__contains__('Pending confirmation in Bitcoin blockchain'):
183             verified = False
184         elif res.__contains__('Success! Bitcoin attests data existed'):
185             verified = True
186 
187         otsevent = {
188             'name': 'VerifyFile',
189             'time': time.time(),
190             'command': 'verify',
191             'message': 'Performed a verification and the result was ' + str(res)
192         }
193 
194         # update the proof
195         otsobj['proof']['verified'] = True
196         otsobj['proof']['verifiedTime'] = getProofTimestamp(res)
197         otsobj['proof']['info'] = res
198         otsobj['proof']['attestationDetail'] = info(getProofPath(otsobj['path']))
199         # add the event
```

```
200         otsobj['events'].append(otsevent)
201
202         # update obj based on _id
203         db.otsfiles.replace_one({"_id": _id}, otsobj)
204
205 def upgrade_timestamps():
206     otsobjs = db.otsfiles.find({"proof.upgraded": False})
207
208     # loop through all the non-upgraded stamps and upgrade them
209     for otsobj in otsobjs:
210         #print(str(otsobj))
211         # get the object id so we can reference it later
212         _id = otsobj['_id']
213
214         logger.info("Found obj ready to upgrade at %s with ID %s" % (getProofPath(otsob
                ↪ j['path']), _id))
215         #print(_id)
216         # perfrom teh upgrade
217         res = upgrade(getProofPath(otsobj['path']))
218
219         logger.info("Found obj with ID %s. Upgrade result %s" % (_id, res))
220
221         #print(str(res))
222         # create an event for the upgrade
223         otsevent = {
224             'name': 'UpgradeFile',
225             'time': time.time(),
226             'command': 'upgrade',
227             'message': 'Performed an upgrade and the result was ' + str(res)
228         }
229
230         # update the proof
231         otsobj['proof']['upgraded'] = res
232         # add the event
233         otsobj['events'].append(otsevent)
234
235         #update obj based on _id
236         db.otsfiles.replace_one({"_id": _id}, otsobj)
237
238
239
240 _starttime = time.time()
241 # Create and save new file
242 logger.info("Create new file started")
```

```
243 create_new_file()
244 logger.info("Create new file ended")
245
246 # See who should be upgraded and upgrade them
247 logger.info("Upgrade timestamps started")
248 upgrade_timestamps()
249 logger.info("Upgrade timestamps ended")
250
251 # See who has been upgraded and verify them
252 logger.info("Verify timestamps started")
253 verify_timestamp()
254 logger.info("Verify timestamps ended")
255 _endtime = time.time()
256
257 logger.info("Process started at %s and ended at %s. Complete run time: %s seconds" % (
    ↪ _starttime, _endtime, str(_endtime - _starttime)))
```

A.2 breakverify.py

Listing A.2: breakverify.py source code

```
1  import os
2  from random import randint
3  import logging
4  from pymongo import MongoClient
5  import subprocess
6  import time
7  from time import mktime
8  from datetime import datetime
9  import re
10 import sys
11 import hashlib
12 import binascii
13
14 logger = logging.getLogger(__name__)
15 logger.setLevel(logging.INFO)
16
17 # create a file handler
18 handler = logging.FileHandler('modify.log')
19 handler.setLevel(logging.INFO)
20
21 # create a logging format
```

```
22 formatter = logging.Formatter('%(message)s')
23 handler.setFormatter(formatter)
24
25 # add the handlers to the logger
26 logger.addHandler(handler)
27
28 testFilePath = os.path.join(os.getcwd(), 'testfilesdev')
29 breaklist = 'modlist.txt'
30
31 def verify(file):
32     start = time.time()
33     res = subprocess.getoutput('/usr/local/bin/ots verify ' + file)
34     stop = time.time()
35     #logger.info("Verify time: %s", str(stop-start))
36     #print(res)
37     return res
38
39 def parseVerify(result):
40         if "Error" in result:
41                 return False
42         elif "File does not match original" in result:
43                 return False
44         elif "Success" in result:
45                 return True
46
47 def getProofPath (file):
48     return file + '.ots'
49
50 def modFile(file):
51     with open(file, 'br+') as f:
52         f.write(b'\x01\x03\x03\x07')
53     #check file size
54     res = str(os.path.getsize(file))
55     return res
56
57 #Load file list
58 with open(breaklist) as f:
59     content = f.readlines()
60
61 content = [x.strip() for x in content]
62
63 for line in content:
64         timelinedata = re.split(r'\t+', line.rstrip('\t'))
65         #print(timelinedata)
```

```
66          name = timelinedata[0]
67          originalSize = timelinedata[1]
68          number = timelinedata[2]
69          originalResult = parseVerify(verify(getProofPath(name)))
70
71          if int(number) % 2 != 0:
72                  #print('Modifying: ' + name)
73                  newsize = modFile(name)
74                  #print(str(newsize) + ':' + originalSize)
75                  result = verify(getProofPath(name))
76                  #print(result)
77                  print(name +','+ str(originalResult) +','+ 'modFile' +','+ str(parseVeri
                        ↪ fy(result)))
78                  logger.info(name +','+ str(originalResult) +','+ 'modFile' +','+ str(par
                        ↪ seVerify(result)))
79          else:
80                  #print('Modifying: ' + getProofPath(name))
81                  newsize = modFile(getProofPath(name))
82                  #print(str(newsize) + ':' + originalSize)
83                  result = verify(getProofPath(name))
84                  #print(result)
85                  print(name +','+ str(originalResult) +','+ 'modStamp' +','+ str(parseVer
                        ↪ ify(result)))
86                  logger.info(name +','+ str(originalResult) +','+ 'modStamp' +','+ str(pa
                        ↪ rseVerify(result)))
```

Appendix B

Other

B.1 Email correspondence with Peter Todd re: OTS timestamp structure.

```
Delivered-To: thomas.weilbach@gmail.com
Received: by 10.176.87.145 with SMTP id x17csp3164088uaa;
        Sun, 1 Oct 2017 19:35:35 -0700 (PDT)
X-Google-Smtp-Source: AOwi7QBj2PhpqeJ0eawxIGDcbDr1TZ+jMdXaqhMPyWcYgejC3sP0UsrEGgm8qu08
    ↪ bpDLCvoW5HmV
X-Received: by 10.223.153.242 with SMTP id y105mr2501401wrb.126.1506911735011;
        Sun, 01 Oct 2017 19:35:35 -0700 (PDT)
ARC-Seal: i=1; a=rsa-sha256; t=1506911734; cv=none;
        d=google.com; s=arc-20160816;
        b=JiQSztkRZ5fX1d1X88DSdusRf1CBRbPHotNAz1arKLK7nAGSpgqla1elQvQvq1NXMf
         SDxOtZOaUUxNGq1QqIDN2YA6KqlsNi8rr3mHwMamii8X3cdlU7uJE9yL9xEyUkxcWv4t
         GML8wAsh6CBybGMkq6LLUFlont241i9jzFyfPj7WLgoiijUjiOozZGEmV51GEkmRFQ8v
         WB7MyR61KohHB33EIcKnz+hU4nC2KHOYd3woW+77FqvWrELsYv42oOoEYcINHq14hsqc
         3ewfW2vx0Zim9hUpTZyUAf27LyDmT9SARYKKk6Headv8YS2yYwpt04rJlm3xqY0BxY7Q
         AGfQ==
ARC-Message-Signature: i=1; a=rsa-sha256; c=relaxed/relaxed; d=google.com; s=ar
    ↪ c-20160816;
        h=user-agent:in-reply-to:content-disposition:mime-version:references
         :message-id:subject:to:from:date:arc-authentication-results;
        bh=UBsONdVST9r8PhgcfhOBcmLZ65I8IdDrPMOwejcLmpw=;
        b=q34zbhGgEf754Ky7BFWZsZ3G/VdbvlHK9zBSfEqcdX7rM9H4Xe8VW9kSLtwfMLkB7Z
         HHEBNbJFPiiJf5er4hA1MgJhPrGvNqSQ/kGqShzaCXcrkkw8CnRod/g5LmQgON4Kakt+
         qAEKhxqLVgKaj8ecgaShPKIKJa++EpXX6xywKDrA5hkYX1bmbqf8cI5wDcBU3O8vRPHt
         KbjOkPr1aXJNUaRa/W7YOgVX1nkQawbwh7h07jGn5ru/jQ4QakxVtZ3sCDAfZkif9ehW
```

```
        PfEfQxvHgR9gNOMTWBRkdl4rwOV/7bhsipFk8IpB4YdbDz1Z3ovAUBcbkf47ZZjsv8td
        KjJQ==
ARC-Authentication-Results: i=1; mx.google.com;
      spf=pass (google.com: domain of pete@petertodd.org designates 62.13.149.75 as pe
          ↪ rmitted sender) smtp.mailfrom=pete@petertodd.org
Return-Path: <pete@petertodd.org>
Received: from outmail149075.authsmtp.net (outmail149075.authsmtp.net. [62.13.149.75])
       by mx.google.com with ESMTPS id e82si7082200wma.192.2017.10.01.19.35.34
       for <thomas.weilbach@gmail.com>
       (version=TLS1_2 cipher=ECDHE-RSA-AES128-GCM-SHA256 bits=128/128);
       Sun, 01 Oct 2017 19:35:34 -0700 (PDT)
Received-SPF: pass (google.com: domain of pete@petertodd.org designates 62.13.149.75 a
    ↪ s permitted sender) client-ip=62.13.149.75;
Authentication-Results: mx.google.com;
      spf=pass (google.com: domain of pete@petertodd.org designates 62.13.149.75 as pe
          ↪ rmitted sender) smtp.mailfrom=pete@petertodd.org
Received: from mail-c245.authsmtp.com (mail-c245.authsmtp.com [62.13.128.245]) by pun
    ↪ t20.authsmtp.com (8.14.2/8.14.2/) with ESMTP id v922ZYZC080484 for <thomas.weil
    ↪ bach@gmail.com>; Mon, 2 Oct 2017 03:35:34 +0100 (BST)
Received: from petertodd.org (ec2-52-5-185-120.compute-1.amazonaws.com [52.5.185.120])
    ↪  (authenticated bits=0) by mail.authsmtp.com (8.14.2/8.14.2/) with ESMTP id
    ↪ v922ZWwm008707 (version=TLSv1/SSLv3 cipher=DHE-RSA-AES256-SHA bits=256 verify=N
    ↪ O) for <thomas.weilbach@gmail.com>; Mon, 2 Oct 2017 03:35:33 +0100 (BST)
Received: from [127.0.0.1] (localhost [127.0.0.1]) by petertodd.org (Postfix) with ESM
    ↪ TPSA id EEF7540101 for <thomas.weilbach@gmail.com>; Mon,
  2 Oct 2017 02:35:31 +0000 (UTC)
Received: by localhost (Postfix, from userid 1000) id 31085205E4; Sun,
  1 Oct 2017 22:35:31 -0400 (EDT)
Date: Sun, 1 Oct 2017 22:35:31 -0400
From: Peter Todd <pete@petertodd.org>
To: Thomas Weilbach <thomas.weilbach@gmail.com>
Subject: Re: Opentimestamps masters research project
Message-ID: <20171002023531.GA10302@savin.petertodd.org>
References: <CAHe_Fo-7qbwXrqoz3qVi_9+9FmOe7WPU=ycVWYuQOX5rFNtyCA@mail.gmail.com>
    ↪ <20170904153357.GX1276@fedora-23-dvm> <CAHe_Fo8JRXO=K9PKAJKY3qcfEc-UPExgWzzsACc
    ↪ Tv_FaBXabxg@mail.gmail.com>
MIME-Version: 1.0
Content-Type: multipart/signed; micalg=pgp-sha256; protocol="application/pgp-signature
    ↪ "; boundary="IJpNTDwzlM2Ie8A6"
Content-Disposition: inline
In-Reply-To: <CAHe_Fo8JRXO=K9PKAJKY3qcfEc-UPExgWzzsACcTv_FaBXabxg@mail.gmail.com>
User-Agent: Mutt/1.5.23 (2014-03-12)
X-Server-Quench: 5d55ec12-a71a-11e7-801f-9cb654bb2504
X-AuthReport-Spam: If SPAM / abuse - report it at: http://www.authsmtp.com/abuse
```

```
X-AuthRoute: OCd2YgOTA1ZNQRgX IjsJECJaVQIpKltL GxAVMQtdJVMSExtc MOJQJ1FTGFIZTFBd HTVUB
    ↪ hpUUkIOCidq aQpQahVdYUBNWg9p UwZMQExXCgRgBAIA BB4BUBxtdksAcXxy DyElA3BbVEN6cgh7
    ↪  RkdcW2hSYGI1O2Ed TRFfdlJJcQIfeQJN alEpSXpeM2MCYigy RVJrYyYLMGcXDzxY RBOMKF8JX
    ↪ U8OBiVO QhkZATg1BgUfSj4+ JgYtKhYEBkETPO4u Wd/M
X-Authentic-SMTP: 61633532353630.1039:706
X-AuthFastPath: 0 (Was 255)
X-AuthSMTP-Origin: 52.5.185.120/25
X-AuthVirus-Status: No virus detected - but ensure you scan with your own anti-virus s
    ↪ ystem.

--IJpNTDwzlM2Ie8A6
Content-Type: text/plain; charset=us-ascii
Content-Disposition: inline
Content-Transfer-Encoding: quoted-printable

On Sun, Oct 01, 2017 at 08:54:38PM +0400, Thomas Weilbach wrote:
> Thanks for the feedback Peter.
>=20
> Work is going well and I=92ve got about 4000 proofs in my dataset. I=92m
> looking forward to analyzing the data and drawing some conclusions.

Great!

> I=92m trying to explain why OTS proofs are saved as binary data blobs but
> can=92t seem to find a suitable justification in any existing documentation=
.
>=20
> Is it simply to prevent issues with encoding and compatibility or is ther=
e
> some other explanation for not saving it in a legible textual format?

You=92re on the right track.

Something to keep in mind with textual formats, is you *always* have to par=
se
the output of the textual format parser itself. For instance, here=92s part o=
f a
timestamp from an earleir 2012(1) version of OpenTimestamps:

    "ops": [
        {
            "Hash": {
                "input": "13249541def3c688e28a32da9a6f39e2c68442db",
                "parents": [
```

```
                    [
                        0,
                        20
                    ]
                ],
                "algorithm": "sha256d",
                "digest": "49bdaf64146928c7ba30e5a28704e0762a37d53236438b4c=
d1d831f0568b8535"
            }
        },
    ]

Is JSON the serialization format for that timestamp? No. The actual format =
is a
subset of the JSON standard, namely exactly what JSON elements are allowed =
in
an OpenTimestamps proof. The code dealing with that format is itself a pars=
er -
a parser that happens to decode JSON objects rather than bytes.

The problem is OTS is security software: unless your code understands the O=
TS
proof its given 100%(2), under *no* circumstances should it consider the pr=
oof
as valid. After all, the OTS proof is untrusted, and provided to your code =
by
the attacker. Furthermore, I want OTS to be usable in consensus-critical
scenarios. That means that we consider it to be a potential security flaw =
if
two different OTS implementations give different results for the same proof=
.

Finally, I want implementing the OTS standard *correctly* to be as easy as
possible. But again, since this is security software - it's not good enough=
 if
you only *think* you implemented the standard correctly, but actually didn'=
t.
Thus priority #1 is to make implementing the standard incorrectly *unknowin=
gly*
to be difficult.

An important subtlety in making a consensus-critical standard easy to use, =
is
there must be a natural upper-bound to how much RAM it takes to parse an OT=
```

```
S
proof; requiring explicit resource tracking and limits to achieve this isn'=
t
doesn't make for an easy to implement standard.

So right away, ignoring even the exact encoding, you'll notice that the OTS
proof format is nearly stateless: the only state an implementation has to k=
eep
track of to parse and verify a proof are the messages at the fork points.

As for the exact encoding, the thing with binary is that it maps directly t=
o
the token-level structure of an OTS proof, with essentially no "glue" in
between. So we've cut the total amount of parsing code required almost in h=
alf.

Secondly, the encoding has almost no redundancy - every single bit in the
serialized proof maps 1-to-1 with something inherent to the mathematical
structure of the proof. This means that if you *incorrectly* parse anything=
,
you will almost certainly hit an obvious error condition and reject the pro=
of
as invalid, rather than accidentally interpret the proof in a different way=
 to
other implementations. For security software, this brittleness is a good th=
ing,
as we want incorrect implementations to fail completely, 100% of the time,
rather than potentially give inaccurate results.

It's true that it's often much easier to debug and learn from human readabl=
e
encodings. But that's why things like the `ots info` command exist. Equally=
,
creating a OTS proof by hand is impossible anyway without *some* tooling, a=
s
humans can't calculate SHA256 digests in their head.

FWIW, the Bitcoin protocol itself uses this exact same strategy, and its tr=
ack
record re: protocol-level security is almost perfect.


BTW, do you mind if I repost this reply to the ots-dev mailing list? It's
useful for everyone to see this. I'm happy to take your name off it if you
```

```
want.

1) git commit 28333aacb9bf4551210d7ea6b94879b703b1ae51, doc/opentimestamps-=
client-v0.1.ots
2) There is a controlled mechanism in the OTS standard to allow implementat=
ions
   to implement less than 100% of the standard: notary attestations. I'm al=
so
   considering adding the notion of backwards compatible upgradable opcodes=
. But
   that's an advanced topic; requiring all validators to uprade at the same=
 time
   is a safe default.

--=20
https://petertodd.org 'peter'[:-1]@petertodd.org

--IJpNTDwzlM2Ie8A6
Content-Type: application/pgp-signature; name="signature.asc"
Content-Description: Digital signature

-----BEGIN PGP SIGNATURE-----

iQEcBAEBCAAGBQJZ0aXuAAoJECSBQD2l8JH7pBoH/3Ri6dyHvtBtgxdb6uo2ie2e
ZUHFIXIZIfaksxnEuFm5uTwbp6cFbxxWJxAr64/0aO0AnvxbUfn8D2ObiGUSHUEs
2WAD0UYswff1JOLIZ4Nfa9RA60vpMTqdOGr0i6HdfU1nsBt9fBh42OkpsR3HeYDo
1E0AisUhIYQobytp4hHzs6p7W5h6D/7zTq8BY9eoO61d0gBhm7XtJZkZyLizsqhH
bR2BmgetwmSs59p0Io/M515XND1D2VOBrSIe2V7yc+BFqnVShjri014YohKWTfeb
z2OTFBSe15/iAK8kKQVzicV79yYHJuthLcXYhVsdYUFpiAxFiH3BAIksYu3dqzQ=
=suQw
-----END PGP SIGNATURE-----

--IJpNTDwzlM2Ie8A6--
```