Imperial College London

Department of Computing

Betting on the Bitcoin Blockchain

Author:
Joshua David Lind

Supervisor:
Dr. William Knottenbelt

June 2015

Submitted in part fulfilment of the requirements for the degree of
Master of Engineering in Computing

Abstract

We present the *Bitcoin Betting Exchange*, the first publicly auditable, entirely anonymous and fully automatic *Bitcoin* betting exchange. By making use of the *Bitcoin Blockchain* and a unique *proof of bet* system, we can minimize the amount of trust required between the betting parties and enforce honest participation. Using the *proof of bet* system we can design a hardware based private key store that acts as an external adjudicator, only ever releasing funds when presented with sufficient evidence.

Existing betting exchanges and websites are rife with criminal activity. The ungoverned and anonymous nature of *Bitcoin* mean that websites can secretly steal users money and users can make false claims against the website. With no proof, it is the word of one against the other.

Our betting exchange presents a unique and novel solution to addressing the issue of *trust in an anonymous environment.* It has been rigorously tested, survived a security audit, numerous attacks online and several vulnerability scanners. An in-depth analysis of the vulnerabilities of our design reveal that under a threat model where an attacker has complete control over the betting exchange, they cannot gain access to the funds held by the exchange, or to the private keys.

Acknowledgements

I would like to express my sincerest thanks to:

- Dr. William Knottenbelt, my supervisor, for providing invaluable guidance throughout my project. Your inspiration, enthusiasm and support have made this project a joy to work on.
- Dr. Sergio Maffeis, my second supervisor. Your feedback and insight have helped shape this project right from the beginning.
- Charlie Hothersall-Thomas, for spending his time penetration testing and analysing our work.
- William Hanbury, for helping propose the project and providing feedback all the way through.
- My family and friends, for their unwavering support throughout my studies.
- God, who has blessed me with the opportunity to study, supported me in the hard times and never given up on me.

This thesis is dedicated to Charles Derek Crisp.

"No one lights a lamp and puts it in a place where it will be hidden, or under a bowl. Instead they put it on its stand, so that those who come in may see the light." Luke 11:33

Contents

1. Introduction

1.1 Motivation

The motivation for this project lies at the intersection between virtual currencies and online betting and gambling. By combining these two unique areas we can construct an entirely *anonymous*, *border-less* and *trust-less* online betting exchange powered by virtual currency. With unlimited transaction amounts, near zero transfer costs and instant payouts, the exchange could offer uncapped competitive odds, with immediate resolution. This would not only free up the global betting marketplace but it would allow anybody with access to the Internet to bet against anybody else, anywhere in the world. By using the *Bitcoin* network as the foundation for a betting system, we can create an anonymous and distributed betting exchange that is globally accessible and completely unconstrained.

Although this idea has already been explored in part by several existing solutions, such as websites like *DirectBet.com* [46] and *BetBTC.co* [11], there are two major issues that have yet to be addressed. The first is *trust in an anonymous environment.* Many, if not all, of the existing solutions require bettors to deposit their bets into an account owned by the betting website or application. The betting website then acts as escrow until the bet has been settled, paying out the winnings to the appropriate user. Because *Bitcoin* is by nature anonymous, the identities of the bettors and the operators who own the sites are never known. Combined with the fact that *Bitcoin* is also ungoverned, it creates opportunity for the various parties to misbehave. For example, a betting website can refuse to pay out winnings to a user, delete any evidence of a bet, and continue operating as though nothing had happened. Likewise, a bettor can falsely claim wrong doing by a betting website in order to ruin their reputation and drive away customers. Both parties have little in way of evidence to prove they have been wronged. Emails and screen-shots do not constitute valid proof because emails can so easily be forged and images manipulated. In the majority of cases it becomes the word of the bettor against the betting service, leaving one with no way to prove the other wrong.

In fact, this problem has become so widespread throughout the *Bitcoin* betting community that there are literally thousands [51] of reports of dishonest operation and foul-play scattered throughout *Bitcoin* forums and various websites online. Hundreds of betting websites and applications have reportedly ceased operation after receiving large deposits from users, and many have incorrectly settled bets or simply refused customer withdrawals. In one case a betting website, *CoinBet.cc*, refused to allow a customer to withdraw around $340,000 dollars [110] worth of accumulated winnings. When the customer complained and petitioned online, the website stopped operating, refused to pay them any money and then later resurfaced online under another name.

The second major issue not yet addressed by any of the existing solutions is that of *private key security and storage*, something that applies to all *Bitcoin* services. To date there have been many serious and alarming cases where a *Bitcoin* application or website has been targeted by an attacker and the private keys held online have been stolen. Because *Bitcoin* is not governed by any central authority, with transactions being irreversible, these victims have been left without anywhere to turn and their funds have permanently been lost. In fact many of the most serious cases have occurred within the last two years alone. *Mt. Gox* for example, a *Bitcoin* exchange that allowed users to trade *Bitcoin* for regular (fiat) currency was reportedly hacked in early 2014 [32]. The result of this was the loss of over 750,000 *Bitcoin*, worth around $500 Million US at the time. Likewise, *BitStamp*, another well known and trusted *Bitcoin* exchange also fell victim to a *hot wallet* attack in early 2015, where private keys stored online were stolen. This resulted in the loss of around 19,000 *Bitcoin*, worth around $5 Million US [93]. The frequency and severity of these attacks highlight the importance of secure and sound key management, with mistakes having disastrous consequences.

In order to overcome these problems and push back against the challenges facing *Bitcoin* betting services, we propose using several of the unique features provided by *Bitcoin* (such as the

Blockchain) to create an entirely public auditing system. Using these features the confirmation of bets, resolution of outcomes and payments of winnings can forever be publicly signed, timestamped and embedded into the network. The auditing system would provide a very open and easy way to verify the honest operation of the parties involved, while at the same time still maintaining complete anonymity. Furthermore, using this proof we can design a hardware based private key store that only ever releases funds when given proof of bet and outcome, without ever exposing the private keys to the betting exchange or outside world.

1.2 Objectives

The goal of this project is to create an online betting exchange in the form of a web application. It is to operate using *Bitcoin* as the primary trade currency, allowing users to place bets on various *sports* events. It should be entirely anonymous, from payments and transactions, to interaction with the website, meaning users should not need to create any form of account or complete any registration process. It should also utilize the trust-less nature of the *Bitcoin* network in order to match bets and propositions without requiring trust between the betting parties.

Furthemore, every stage of the betting process should be entirely transparent and auditable. This means that at every step in the sequence of events the actions taken by the participants should be publicly traceable and verifiable, without compromising anonymity. This means that any claims of dishonest operation or disputes from either party can be backed up with verifiable and concrete proof. This will help to ensure the honest operation of the betting exchange as well as the bettors involved, minimizing the amount of trust required between the participants. In addition to this, bets and wagers should be automatically settled as efficiently as possible in order to minimize settlement waiting times.

Specific attention should also be given to the security of the private key store and overall architecture of the design, taking into account the threats that arise from storing key data online in *hot wallet* scenarios.

1.3 Contributions

The main contributions provided by this thesis are summarized below:

- A web application that allows users to place uncapped and unconstrained bets on real sports events and markets in real time. The application operates anonymously, requiring no personal information or user sign up, and runs over an encrypted connection.
- A proof of bet system that uses the *Bitcoin Blockchain* as a way to sign, timestamp and publicly announce user bets without compromising anonymity. This provides a way for users to concretely prove and verify their bets.
- A proof of outcome system that allows trusted sports authorities to publicly and verifiably announce the outcome of various events and markets. This prevents the betting exchange from being able to incorrectly settle any bet without consequence.
- A verification check that allows third parties to verify the correct resolution and settlement of all bets. This check is constructed using the proof of bet and outcome systems listed above in combination with the public transaction information embedded in the *Blockchain.*
- The design and implementation of a hardware based, smart, private key store that makes use of the above verification checks to sign payments and settle bets. The key store performs these actions without ever exposing the private keys to the betting exchange or to the outside world. This means that under a threat model where an adversary has control over the web application, the private keys and funds for the exchange still remain intact.

In this thesis we present the *Bitcoin Betting Exchange*, a publicly auditable, anonymous and fully automatic betting exchange that operates on the *Bitcoin* network. The *Bitcoin Betting Exchange* and the private key store are both fully operational and can be accessed online at: `https://bitcoin-betting.herokuapp.com`.

2. Background

In this chapter we provide the background required to understand betting in a digital currency environment.

We begin, first, by providing a general introduction to digital and virtual currencies and continue thereafter by exploring the technical aspects required to understanding *Bitcoin*. We then provide a brief introduction to online betting and gambling and describe the background associated with sportsbooks and betting exchanges. We conclude finally by discussing the existing solutions and related work in this area.

2.1 Digital and Virtual Currencies

2.1.1 Overview

Over the past 25 years the Internet has given rise to countless virtual and digital currencies. Ever since the early 1990's the idea of a border-less, tax-free and ungoverned currency has led to the rise and fall of many different electronic and digital concepts. From *E-gold* [69], a digital currency founded in 1996 that was backed by gold, to *Beenz*[38], a virtual currency that was tradable for goods in online marketplaces, electronic currencies have taken root in the freedom and openness that the Internet provides.

Today, there are over 80 [79] different digital and virtual currencies available online. Each currency with its own unique design and purpose. In order for us to identify and analyse the different currencies it is important to define exactly what we mean when we use the terms *digital* and *virtual* currencies. First and foremost, *digital* currencies are currencies that exist only in a digital, binary format. 1s and 0s. All transactions occur in the digital realm, and these currencies are never physical embodied. Meaning, all money is created and stored electronically. The primary difference therefore, between *digital* and *physical* currencies are that *digital* currencies are never physically produced. You cannot hold them in your hand or put them in your pocket. *E-gold* for instance is a *digital* currency. It exists only in an electronic form. The British Pound however is a *physical* currency. It can actually be held in your hand. Take a £5 note for instance, or a £1 coin. Although this difference is somewhat clear, the distinction between *digital* currencies and *physical* currencies can become blurred when you consider the fact that there isn't actually enough tangible, physical currency to supply the worlds demand. Meaning, that if everyone were to go to their bank today and fully withdraw their accounts, there wouldn't exist enough physical coins and notes to satisfy the request [129]. Be that what it may, *physical* currencies can be held in your hand, *digital* currencies cannot.

What then is a *virtual* currency? A *virtual* currency is a type of *digital* currency. It exists primarily in a *virtual* world. An example of this might be *gold coins* in the massive multiplayer, online role-playing game *World of Warcraft*. These currencies are limited to the virtual worlds in which they exist. As such, *Bitcoin* is often incorrectly referred to as a *virtual* currency. In fact, the primary motivation for *Bitcoin* is to provide a currency that can be used and traded in the real world, such as to exchange for U.S Dollars, or to purchase a cup of coffee. *Bitcoin* is therefore a *digital* currency. More specifically it is a *digital cryptocurrency*, a currency built using cryptography as an implementation for managing transactions and generating new coins.

With so many of these different *digital* currencies available, why choose to pay particular attention to *Bitcoin*? Well up until the time it was first proposed, in 2008, every *digital* currency was built using a single, centralized clearinghouse, or authority. This authority was responsible for managing account balances and money transfers, similarly to how a bank operates. The problem with this design was that it introduced a single point of failure. If the authority was attacked or shut down, the currency immediately became worthless. This was a fundamental problem for the majority of currencies, and as a result, many of them collapsed due to targeting by concerned governments, cyber-criminals, legislative action or liquidation of their parent company.

In 2008 however, all this changed. A whitepaper entitled *Bitcoin: A Peer-to-Peer Electronic Cash System* [80] was published online. Released under the pseudonym *Satoshi Nakamoto*, the paper described the worlds first decentralized digital currency. By making use of a distributed peer to peer network and a public transaction ledger, the paper resulted in the worlds first decentralized crypto-currency, the first *digital* currency to provide a solution to the central authority problem. It replaced the single centralized authority with a *distributed* network of many authorities and provided a way to achieve consensus among them. In doing this, there was no longer a single point of failure. In order for an attacker to render the currency worthless it would now need to corrupt at least a majority of the distributed network. Something that is incredibly unlikely when you consider that the number of active authorities in the *Bitcoin* network is in the order of 10s of thousands [88].

Over the past 7 years, *Bitcoin* has rapidly gained popularity and widespread acceptance. To date there have been over 60 million transactions [26] in the *Bitcoin* network, resulting in an estimated total market capitalization of 3.3 billion U.S dollars [25]. But, despite its immediate purpose, *Bitcoin* has also become the source of innovation for several different reasons. Its open-source nature and unique implementation features, such as the *Blockchain*, make it highly attractive to extension. This is true not only for the development of customized alternate currencies, such as *Bytecoin* [31] and *Litecoin* [73], but also for completely different applications altogether, such as smart contracts, proof of existence and entirely new technology protocols, often referred to as *Bitcoin 2.0* [83].

2.1.2 Bitcoin

As previously mentioned *Bitcoin* is a *digital cryptocurrency.* This means that at its heart it uses *cryptography* for transaction validation, managing security and ensuring mathematically fair generation and distribution of new coins. *Cryptography* is the study of techniques to allow secure and secret communication between parties, and is a branch of mathematics and computer science.

In order to understand how *Bitcoin* works, it is important to clarify exactly how an amount of money is represented in *Bitcoin.* With traditional fiat currencies, such as the British Pound for example, an amount of money is represented by the summation of physical notes and coins. For example, if I was to tell you that I had £100, I would show you two £50 notes. This is not so with *Bitcoin.* Rather than have some representation of a coin, or a note, *Bitcoin* uses a transaction ledger, a long chain of transactions effectively proving the derivation of any *Bitcoin* amount. This means that if I wanted to show you that I had 100 *Bitcoins* (*BTC* for short) I would instead show you all the transactions of amounts sent me to, to prove to you how I got it. For example, I would show you the transactions where my dad sent me 50 *BTC*, my mom sent me 30 *BTC* and my friend sent me 20 *BTC.* All combining to make 100 *BTC.* Of course, in order for you to verify that my dad could actually send me 50 *BTC*, you would need to verify that he had that amount of *BTC* in the first place. So you would also need to see the transactions making up his 50 *BTCs.* He would need to show you the transactions sent to him to make up this amount, and you would repeat this process all the way down the transaction chain, until you reached the bottom of the chain. The bottom of the chain might be where some sort of trusted authority issued the money (such as the government), or it may be a proof of how the money was initially generated.

Although this may seem strange at first, it is effectively what banks and many other companies have to do internally to keep track of finances. For example, what would happen if someone hacked into the database storing their bank account balance and changed it from £100 to £1,000,000? The transactions are used as a means for protecting against this and allowing financial auditing to take place.

Figure 1 represents diagrammatically how this might work.

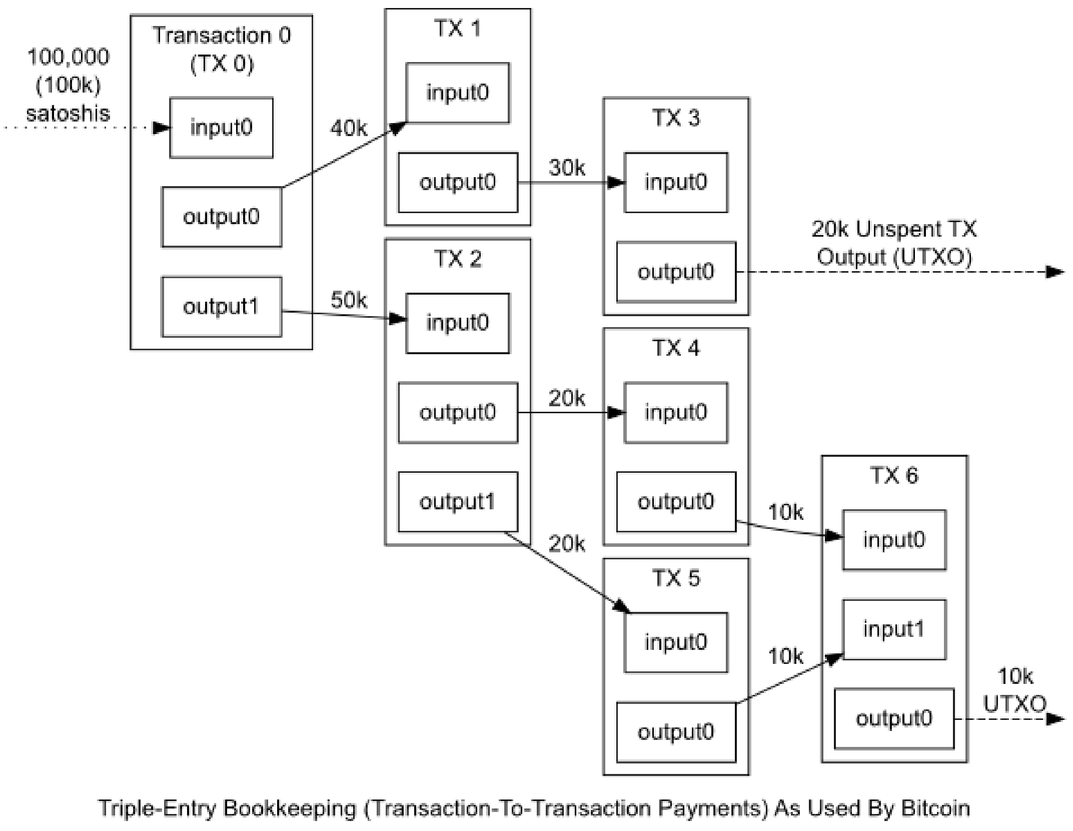


Figure 1: Transaction to Transaction Payments [20]

Note that the diagram uses several phrases that may appear confusing at first. The term *satoshi* in the diagram is the smallest fraction of *Bitcoin* that can be spent. Just like £1 can be split into 100 pence, with 1 pence being the smallest denomination, a *satoshi* is the smallest spendable denomination of *Bitcoin*, 0.00000001 *BTC*, or 10 nano *Bitcoins*. Likewise, *TX* in the diagram stands for transaction and *UTXO* stands for unspent transaction output, an amount of money that has yet to be spent by another transaction.

In the diagram we have 7 different transactions represented as 7 large boxes. Each is labelled *TX 0* to *TX 6*. As mentioned above, in order to spend an amount of money, you need to prove that you own that amount by providing a set of transactions that together sum to atleast the amount that you want to spend. Therefore, in the diagram every transaction has a set of input transactions. For example *TX 6* at the end of the chain, uses *TX 4* and *TX 5* as inputs. The total spend output of *TX 6* is 20k, or 20,000 *satoshi*, which is less than the total input of *TX 6*, which is 30,000 *satoshi*. In a similar manner, transactions can have multiple outputs. For instance if you want to split some amount of money between different accounts. This is what *TX 0* does. It pays *40k*, or 40,000 *satoshi*, to one account, and *50k*, 50,000 *satoshi*, to another account. This is also useful for generating change from a transaction. If Alice wants to send 1 *BTC* to Bob, but only has a single transaction sent to her of 1.5 *BTC*, she can create a transaction with two outputs, one of 1 *BTC* to Bob and another of 0.5 *BTC* back to herself.

You may have noticed in the diagram that the sum of the inputs is 10k greater than the sum of the outputs, such as *TX 4* for example, where we have a single input of 20k, but only a single output of 10k. The 10k difference between these amounts in the example is referred to as the *transaction fee*. This fee is paid to the *Bitcoin* network in order that the transaction would be validated by the network. This may seem confusing at the moment, however, it will make more sense when we examine the processes involved with broadcasting these transactions later on. The term unspent transaction output, as with *output 0* in *TX 6* simply means that this transaction has yet to be spent or referred to in another transaction.

The *Bitcoin* network uses a global, public transaction ledger, similar to the diagram, to store every transaction that has ever been made on the network. This ledger is publicly available and a copy of it is held by every participant, or authority in the network. Whenever any transaction takes place on the network, for example *Alice* sending 1 *BTC* to *Bob*, that transaction is broadcast on the entire network and every participant updates its copy of the ledger. This is why *Bitcoin* is referred to as a decentralized currency, as each participant effectively acts as a single authority.

At first this is surprising, because it effectively means that everybody can see everybody else's transactions, e.g. *Alice* sent *Bob* 1 *BTC*. What's more, everybody also knows everybody else's total bank balance, simply by calculating every incoming and every outgoing transaction for that account. How then can *Bitcoin* be anonymous? The clarification here is that when *Alice* sends *Bob* 1 *BTC*, her and Bob's names are not associated with the transaction. Instead *Alice* uses her *public key* to send the amount to *Bob's public key*. These keys are elements of *public key cryptography*, an area of cryptography that uses practically irreversible functions to encrypt secret messages and create digital signatures. A *public* key is simply a long alphanumeric string of characters, with a related *private* key, together making a *key-pair*. The idea is that a *public* key can be shared and seen by everyone, but the corresponding *private* key is kept secret. It is this *private* key that allows the holder of the *key-pair* to identify itself, through the use of a digital signature. Because these keys are nothing more than alphanumeric characters, nobody can identify *Alice* or *Bob* from their public keys. So although you can see a transaction from 1 public key to another, you have no idea who owns those keys. This is similar to if you intercepted an email sent to an address `2D4nfD434D934@email.com`, even if you knew what the email said, you have no idea about who owns the account.

The way that a *Bitcoin* bank account is therefore represented in the network is as a *private-public key-pair*. Just like a password, whoever has access to the *private* key has access to the funds in the account, and can make payments to other accounts via their *public* keys. The relationships between these key-pairs is what makes them interesting, i.e. it is easy to derive a *public* key from a *private* key, but not the other way round. Meaning that even though your *public* key can be seen by everyone it is practically infeasible for them to try to derive your *private* key from it. Therefore, by sending money between *public* keys, the identities of the people who own those keys are kept secret.

2.1.3 Block Mining

Going back to our example where *Alice* sends 1 *BTC* to Bob, one might ask exactly how new transactions are added to the transaction ledger. Each participant or node in the network stores the transaction ledger as a linked chain of blocks, called the *Bitcoin Blockchain*. Each *block* in the chain stores a set of transactions that have been validated. When a new block is added to the chain at any node, that block is broadcast to all other nodes in the network, who in turn check its validity and then update their own chain with that new block. In order for any node to add a new block to the blockchain, a large amount of computation or work needs to be performed. This computation is required to prove the validity of the transactions in the block. The key element here is that it requires considerable computation for any node to prove a block, but once that block has been proved, it is fairly easy for any other node to check that proof. The process of adding a new block to the blockchain is called *mining*.

The easiest way to think about it is to picture this process as a game. Every node in the network has a replicated copy of the transaction ledger, stored as a blockchain. As time passes by, new transactions are broadcast to the nodes in the network, such as when *Alice* sends some money to *Bob*, or *Bob* sends some money to *Eve*. As all of these transactions occur, each of the nodes in the network is competing against each other to try to find a solution to a specific problem that will allow them to generate a new block. Whichever node finds the solution first broadcasts it to the rest of the network, who in turn check to see if the solution is correct, and if so, updates their own blockchain with the newly provided block. The node who found the solution first is declared the winner and the game repeats with the next set of transactions that occur.

Because the solution is difficult to find and requires a large amount of computation, why would these nodes spend their time trying to solve it? Well just like a game, the winner receives a prize. In this case, whoever solves the problem first is paid two separate amounts of Bitcoin. The first amount is the sum of all the *transaction fees* of the transactions included in the new block. These fees are the differences between the inputs of a transaction, and the outputs. For example the missing 10,000 *satoshis* for each transaction that we identified in figure 1. These fees will act as an incentive for the miners and because the miner can choose which transactions to include in the new block, it will naturally choose the transactions paying the highest transaction fees.

The second amount of money the winning miner receives is a pre-defined amount of new bitcoin. Just like when a central bank prints new coins and notes, the *Bitcoin* network creates new bitcoins whenever a block is added to the blockchain. The amount created for each new block depends on the number of blocks in the blockchain. At present this amount is *25 BTC*, but it halves every 210,000 blocks. By halving the amount of newly generated bitcoins at these intervals, it introduces a fundamental limit in the total number of bitcoins that will ever be produced, a limit of around *21 million* bitcoins [122]. This imposed limit therefore makes *Bitcoin* a *deflationary* currency.

The process of mining can therefore be considered a game of rounds, where at each round a new block is generated, validating a set of transactions and rewarding the winner. Because of the competitive nature of this game, you might try to think of ways of beating it, or increasing your chances of winning. One way might be to try and throw more and more computing power at the problem. This is where the ingenuity of the *Bitcoin* network begins to reveal itself. The difficulty of the challenge to solve for each block is dynamic and continuously adapts to the power of the miners. The network tries to keep the difficulty of the problem at a level that limits the generation of new blocks to around 1 every 10 minutes. This allows it to control the rate of generation of new coins and the growth of the blockchain. It does this by monitoring the speed at which new blocks are generated and every 2016 blocks either increases or decreases the difficulty depending on how the miners coped [125].

The challenge that the miners need to solve in order to generate a new block is to repeatedly *hash* the header of the potential block with a random number until the generated answer is less than or equal to some *target* shared by the network [125]. *Hashing* is a cryptographic 1 way function that is designed to be irreversible. The idea is to take some input, a string or a message, and apply some function to that input in order to generate a fixed length output string or *hash*. The function is designed in such a way that you cannot derive the original message or input from the output *hash*. The hashing algorithm used by the bitcoin network is *SHA256*, an algorithm that given any length input string produces a hash of length 256 characters. The network can vary the difficulty of the challenge by increasing or decreasing the network *target*. One great property of this challenge is that, arguably, there is no pre-defined method for selecting which random number to use when hashing the head of the block and so finding a solution to the problem is considered luck. Therefore the only real approach to solving the challenge is to repeatedly calculate a new hash each time, brute forcing the possible inputs, until you get lucky.

At the current time, it is estimated that the combined power of the network, measured in hashes computed per second, is around 3.5×10^{14}. This illustrates just how powerful the network has grown to become. Figure 2 shows the networks combined power in billions of hashes per second, called the *hash-rate*, plotted against time for the past 2 years. From this graph we can see that the hash-rate of the combined *Bitcoin* network has nearly quadrupled in the past 6 months.

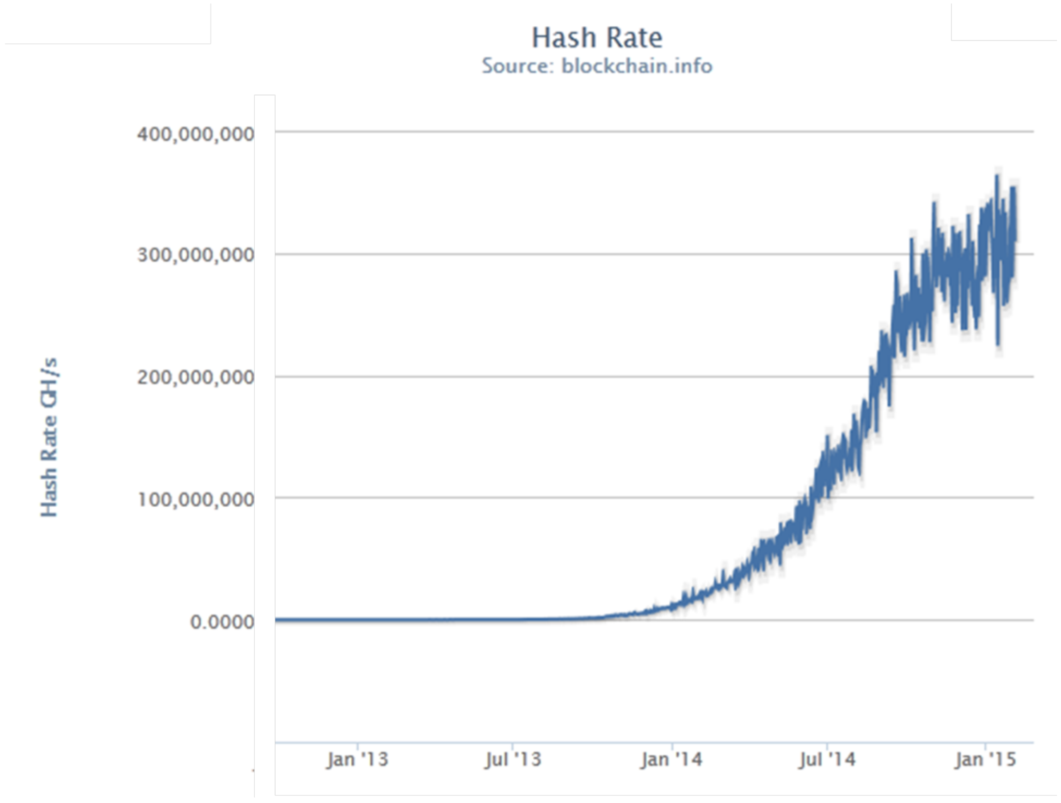


Figure 2: The combined power of the Bitcoin network measured in hashes per second [24]

When we consider the nature of the challenge that miners are required to solve in order to mine a block, we see that the challenge to solve is also ideal when we look at the required number of transactions per block. As the network enforces no minimum or maximum number of required transactions per mined block, just some spatial requirements, one might think that the challenge would be easier to solve with fewer transactions in the block. This however is not the case. As mentioned above, the challenge is to hash the *block header* with some random number. Due to the way in which the block header is derived, through the use of a structure called a *merkel tree*, the difficulty of the problem does not depend on the number of transactions per block. Because of this, it would make sense for miners to want to include as many transactions per block as possible, because of the additional *transaction fees* at no increased difficulty. Likewise, during its infancy, when there were not that many transactions occuring regularly on the network, the ability to mine blocks without needing transactions was an incentive for miners to invest their computation power in order to keep the network running.

2.1.4 The Blockchain

With the public transaction ledger replicated across every node in the network, one might ask what happens in the case that two nodes disagree. For example, if two nodes successfully mine different blocks at exactly the same time and broadcast them across the network. In order to answer that it is necessary to understand exactly how the chain is built and what we mean we say two blocks are *chained* together.

As mentioned in the previous section, in order to solve a block, a miner needs to solve the challenge of finding an appropriate hash for the header of a block. The header of a block is dependent on the transactions in the block, via a *merkel tree*, as well as on the hash of the header of the previous block. To illustrate this point take a look at figure 3.

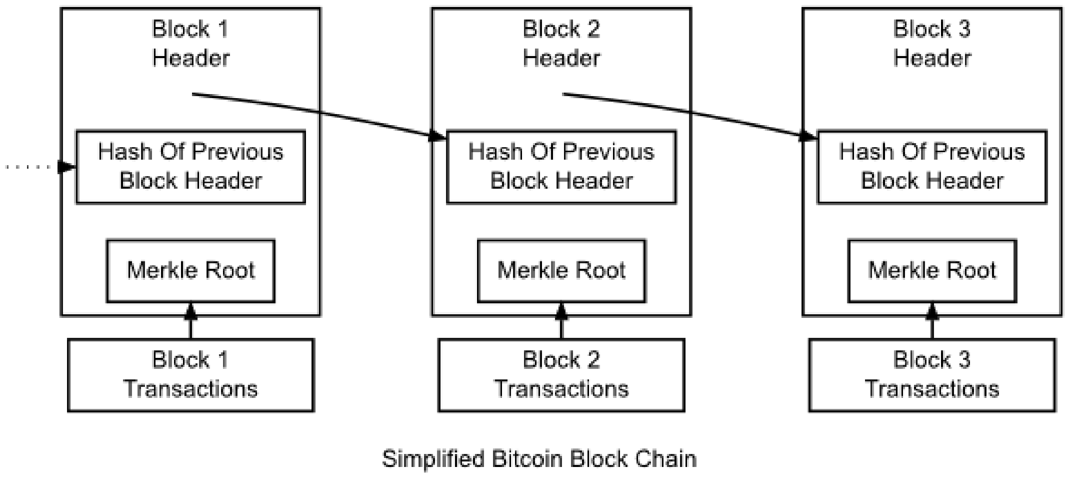


Figure 3: Block chaining [19]

In this diagram we can see three block headers represented by big boxes labelled *Block 1 Header*, *Block 2 Header* and *Block 3 Header*. These block headers correspond to blocks in the blockchain. If we look at the *Block 2 Header*, we can see that it depends on the hash of the *Block 1 Header*, i.e. the solution to the challenge for block 1. It also depends on the root of the *merkel tree* representing the transactions in block 2. This structure is identical for all the blocks in the block chain right down to the beginning of the chain where the very first block was created, called the *genesis* block. The way in which these blocks are chained together has some important and surprising implications. The first of which is that the solution to each block relies on the solution to the previous block, meaning that it is impossible to attempt to solve blocks out of order, i.e. in order to build and solve the second block, you need to have already solved the first block. This requirement enforces the sequential generation of blocks.

The implications of sequential block generation are actually very strong. We know that the time taken on average to solve a single block is around 10 minutes, it therefore means that the longer the block chain, the more time and computational resource that has been invested into it. What's more, this dependency also means that the change or corruption of any information in a block in the chain, will result in the invalidation of that block, as well as **every block that comes after it**.

For example, imagine if someone was trying to attack the blockchain shown in figure 3. What they wanted to do was remove the details of a single transaction from *block 1*, i.e. remove the transaction where *Alice* sends *Bob* 1 *BTC* from the set of transactions in the block. If they applied this change, they would need to update the *merkel root* in the block, as the *merkel tree* is calculated based upon the transactions in the block. Likewise, because that would change the contents of the first block header itself, they would need to re-solve the challenge issued by the network, as the output hash would change because the input block header changed. This would change the solution of the first block. Now because the solution to the second block in the chain depends on the solution of the first block, the solution to the second block would also need to be recalculated, thereby affecting the solution to the third block, and so on. This cascade causes every single block after the modified block to change.

This is a powerful property, because it means that if you insert a piece of information into a block in the blockchain, as you add subsequent blocks to that chain, you increase the amount of work that would need to be done in order for someone to corrupt that piece of information (i.e. they would need to recalculate the entire chain after that block). This feature is often called *proof of work* and is the way in which the network generates trust. We will revisit this property later.

This mechanism can also be used to solve disputes among nodes and achieve consensus as to the order and outcome of operations. If we go back to the question raised earlier about what might happen if two nodes were to disagree, we can see how this comes into play. For instance, imagine that two nodes find a solution to the challenge for the next block at the same time. These nodes then broadcast their solutions to the rest of the network. Who should the network side with? This problem is often referred to as *blockchain forking.* Figure 4 illustrates the problem. We have two valid solutions to the next block after *Block 2*, *Block 3.1* and *3.2.* Both of these solutions were found at the same time and are valid.

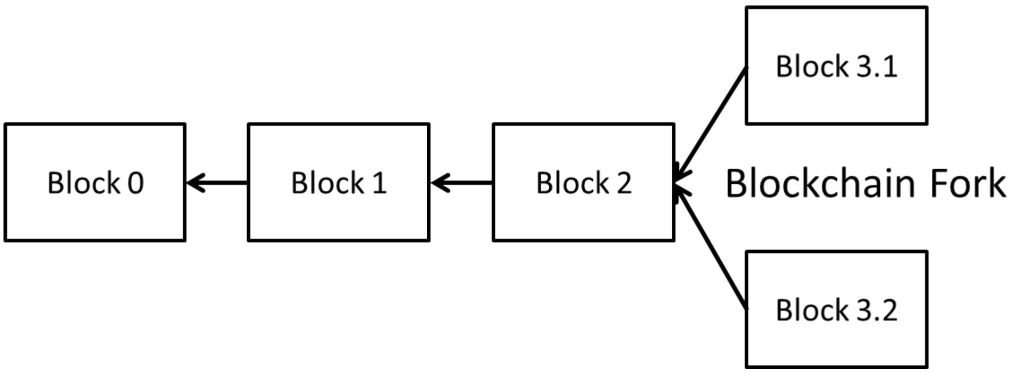


Figure 4: Block forking

What will happen is the two chains will develop side by side simultaneously. Each node will keep a copy of both chains until one of the chains becomes longer than the other, at which point the longer chain will be accepted as the real one, and the smaller chain will be discarded (or *orphaned*). Why will the network select the longer chain as the correct one? As discussed, the longer the chain, the more time and computational resource invested into it, meaning a greater proof of work, and hence more trust.

Furthermore, this property also protects the entire network from corruption. If an attacker wanted to create its own fake chain fork and trick the network into accepting it, this chain would have to be longer than the current one. As such, the attacker would need to have more computational power than the entire network combined to grow its own chain faster than the one produced by the network. This is extremely unlikely considering the size of the network and it has been argued that if profit was the main focus of the attacker, they would actually gain more to follow the rules than to attack it [127]. What's more, even if an attacker could hold the majority of the power, the worst they would be able to do is to reverse their own transactions, effectively issuing what is known as a *double spend attack.*

A *double spend attack* is when an attacker is effectively able to spend the same transaction twice. It is best illustrated with an example. Imagine *Alice* wants to buy a cup of coffee from *Bob* using bitcoin. When she pays *Bob*, she broadcasts a transaction on the network that sends some amount of *BTC* to the address that *Bob* requested. *Bob* sees that the transaction has been broadcast, makes *Alice* her cup of coffee and *Alice* leaves the shop. Note that even though *Bob* saw the transaction was broadcast, this transaction was unconfirmed, meaning that it had yet to be added to a new block in the blockchain by a miner. Because *Bob* didn't want to have to wait 10 minutes for the block to be mined, he gave *Alice* her cup of coffee straight away, trusting that the transaction would soon be mined into the chain. The problem with this is that, if *Alice* was able to mine blocks more quickly than the rest of the network, she could immediately create a different transaction, sending the amount of money she sent to *Bob* back to herself, or to someone else. She would then add that transaction to a new block, broadcast the solution, and everyone would add that block to their chains. However, when the transaction that *Alice* originally sent to *Bob* was to be verified and added into a new block, the network would see that the transactions referenced as input to that payment had already been spent in the block that *Alice* just recently added. It would therefore ignore the transaction as invalid and *Bob* wouldn't receive his money.

What went wrong? The problem with this situation is a result of the way in which the network solves disagreement. It effectively says that the longest chain, or the chain with the highest *proof of work* is correct. If *Alice* has the ability to mine a block before the rest of the network, she could spend the transactions referenced in the payment she sent to *Bob* before the payment is confirmed. One might intuitively suggest that *Bob* should simply wait until *Alice's* transaction is confirmed by the network before giving her a cup of coffee. However this is not good enough because *Alice*, having a majority of the network power, could fork the blockchain just before the point where the network accepted *Bob's* payment, generate two new blocks invalidating that payment, and broadcast the longer chain to the network. Theoretically there is no strict limit as to when it becomes impossible for *Alice* to generate a longer chain. There are however practical limits and trade-offs. The practical cost of having a majority of network power completely dwarfs the costs saved by stealing a cup of coffee. Likewise, even if *Alice* didn't have a majority of the network power, the practical chance of her getting lucky and being able to mine a new invalidating block before the rest of the network is so low that *Bob's* decision not wait for the confirmation is the practical and correct thing to do.

The story changes however, if instead of *Alice* purchasing a cup of coffee from *Bob*, she buys a new *Ferrari*. Now the cost trade-off is different, and it would be wise for *Bob* to wait some amount of time for new blocks to be added to the chain, re-confirming the transaction. A natural question to ask is how long should he wait in this case? Although there is no concrete answer it has been recommended that waiting for *6 block confirmations* (or 5 new blocks on-top of the particular transaction block). This reduces the chance of an attacker successfully performing a *double spend attack* to less than *1%* if they had more than *10%* of the *hash rate* or computing power of the network [119]. With around 1 block being mined every 10 minutes, this would require a wait time of 1 hour, which is more than reasonable considering that banks may take more than 24 hours for a payment to be processed. In fact, the *double spend attack* is only one particular example under a class of attacks called *consensus attacks.*

2.1.5 Transaction and Scripts

Up until now, whenever we have referred to a *Bitcoin* transaction we have assumed a construct that sends money from one *public key* to another. In reality this is only one type of transaction supported by the network and for the sake of the *Alice-Bob* example is a simplification of the protocol. The *Bitcoin* network can actually support several different and more complicated types of transactions, allowing much more than direct money transfers from one person to another.

The first thing we need to understand in order to visualise this is the notion of a *bitcoin address.* A bitcoin address, like a *public key* is a string of alphanumeric characters that can be publicly distributed to allow the transfer of funds. What's important to note is that in the case of *Alice* sending money to *Bob*, the destination or output address that Alice uses is derived from Bob's *public key*, it is not just Bob's *public key.* The bitcoin destination address and Bob's public key **are not the same thing**. In fact, the address can be derived directly from the public key by using a series of hash functions, such as `SHA256` and `RIPEMD` [6]. In the case that Alice wants to send money directly to Bob, the address that Alice uses as the bitcoin output address is calculated directly as:

```
Bitcoin address = RIPEMD160(SHA256(Bob's public key))
```

The resulting address is often displayed to users under a special encoding called *Base58Check encoding* to make the output more readable and to protect against errors via a *check-sum.* This type of transaction is typically referred to as a *Pay-to-Public-Key-Hash* or P2PKH for short, as the amount in the transaction is being paid directly to the hash of the public key. As discussed earlier, in order to display ownership of a public key, it is sufficient to sign something with the corresponding *private key.* Therefore, in this example, if somebody wanted to generate a new transaction referencing Alice's payment to Bob as an input, they would need to be able to sign the new transaction using Bob's private key, which only Bob has.

Whenever a transaction is being validated for use in a new block, the mining node will execute what is a called a *script*, or a sequence of instructions, in order to check that the inputs of any transaction can actually be spent. In order to do this, the mining node executes two scripts, the *locking* script, and the *unlocking* script. The *locking* script is a form of lock placed on the outputs of any transaction. In order for someone to spend those transactions (i.e. reference them as inputs to a new transaction), they need to meet the requirements set by the locking scripts. Likewise, an *unlocking* script is a script placed on the inputs of a transaction. These scripts are run in order to unlock the *locking* scripts placed on these inputs by the previous transactions.

The easiest way to think about it is like a puzzle. In order to spend any transaction, you need to solve the puzzle placed on the transaction by that transactions *locking* script. When you have found a solution to the puzzle you enter it by writing the steps in the *unlocking* script. This is so that when the transactions are being validated by a mining node, these scripts when executed together will solve the puzzle and at the end of the execution will allow the amount to be spent. Successfully unlocking the script occurs when after execution of both scripts together, the result is `1`, representing *TRUE*.

The script language that is used by *Bitcoin* is a very simple stack-based language. It was deliberately designed not to be *Turing complete*, meaning that not every program can be represented or built using it. The reason for this is to protect the network from potential attacks such as *infinite loops* or *denial of service*, by using very long, complex programs in an attempt to slow the network down.

To provide concrete script examples, imagine *Alice* constructs a transaction with a *locking* script that simply returns `1` or *TRUE* always. In this case, anybody can spend the transaction because the answer to the puzzle is trivial, an empty *unlocking* script will do, because when executed together the result will be 1 and thus the transaction will be valid. Likewise *Alice* could also construct a transaction with a *locking* script that simply returns `0` in all cases. This transaction will be *provably unspendable*, because regardless of any *unlocking* script, there is no way that the combined scripts can result in *TRUE*.

A more common example is that of a *Pay-to-Public-Key-Hash* transaction. An illustration of this transaction script can be seen in figure 5 below. Let's assume that *Alice* has sent 1 *BTC* to *Bob* and *Bob* has created a new transaction in order to try and spend *Alice's* payment. This new transaction will need to unlock the *locking* script that *Alice* placed on her payment. In order to do this *Bob* generates an *unlocking* script that he thinks will do the job. In figure 5 you can see *Bob's unlocking script* on the left and *Alice's locking script* following it on the right. When a mining node attempts to verify the new transction that *Bob* has made, it will execute the scripts together from left to right.

Remembering that the scripting language is stacked based and that in order for verification to succeed, the result of the entire script must be *TRUE*, the flow of execution will go as follows: First, *Bob's signature* will pushed onto the stack, this then followed by his *public key*. The *OP_DUP* instruction will then be executed, duplicating whatever is at the top of the stack. In this case, it will pop *Bob's public key* off the top of the stack, make a copy of it and then push them both back onto the stack. The *OP_HASH160* instruction will then pop the top of the stack, *Bob's public key* copy and hash it, pushing the resulting hash back onto the top of the stack. Next, *Bob's public key hash* as provided by *Alice* will be pushed onto the top of the stack. The *OP_EQUALVERIFY* operator will then execute, popping the top two elements off the stack and comparing them. In this case these will be *Alice's* provided version of *Bob's public key hash*, and the calculated version of *Bob's public key hash*. These values will be equal, as we assume nothing has gone wrong, and the *OP_EQUALVERIFY* check will succeed. Finally, the *OP_CHECKSIG* operator will execute, popping the very last two values off the stack, *Bob's original public key* and his *signature* and attempt to verify them. This verification will succeed, and a resultant *1* will be pushed onto the stack. Now that the script has finished being executed, the mining node will pop the top value of the stack, in this case the *1* we just pushed, and check if the final result is *TRUE*. This will succeed as *1* and *TRUE* are the same, and therefore the mining node will accept the transaction as valid.

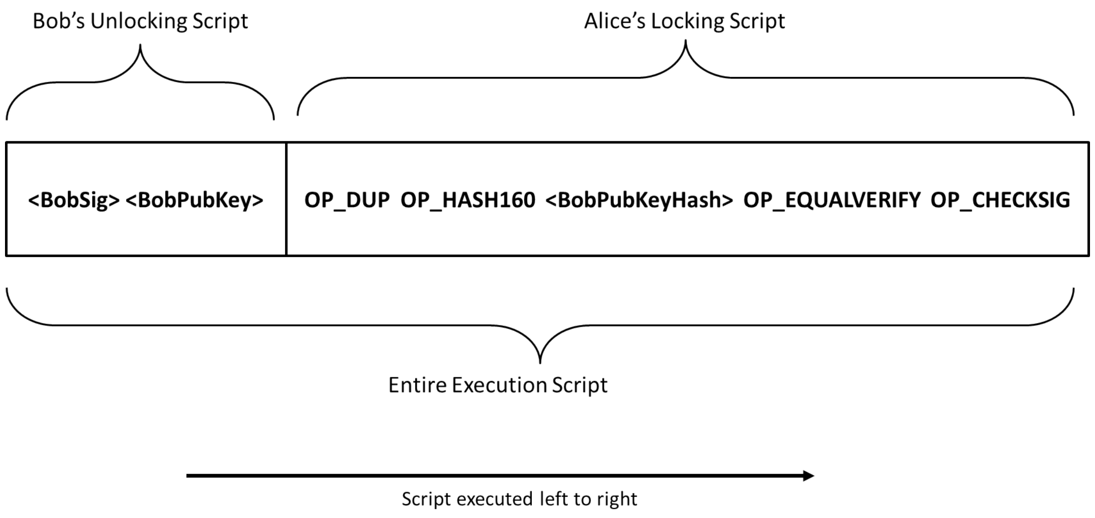

Bob's Unlocking Script
Alice's Locking Script
<BobSig> <BobPubKey>
OP_DUP OP_HASH160 <BobPubKeyHash> OP_EQUALVERIFY OP_CHECKSIG
Entire Execution Script
Script executed left to right

Figure 5: Bitcoin Pay-to-Public-Key-Hash Script

At present, the developers of *Bitcoin* have limited the types of supported transactions on the network to 5 specific constructs. Although this may only be a temporary limitation, these supported constructs are referred to as *standard* transactions and are currently supported by all mining nodes in the network. The 5 supported constructs are *Pay-To-Public-Key-Hash*, *Pay-To-Public-Key*, *Multi-Signature*, *Pay-To-Script-Hash* and *Data-Output*. Although the other script types are technically *unsupported*, there are mining nodes who will take these scripts to be valid when generating new blocks. At the current time however, these transactions are under no guarantee to be mined. And as such, we will only discuss the 5 supported types below.

As we have previously seen, the *Pay-To-Public-Key-Hash* transaction type allows a payment to occur from one party to the public key hash of another. In order for that transaction to be spent by the receiving party, they need to provide a signature using the corresponding private key from the key-pair. Likewise, a *Pay-To-Public-Key* transaction allows the transfer of funds from one party directly to the public key of another. This is effectively a simpler form of the *Pay-To-Public-Key-Hash* transaction and is more commonly associated with older mining nodes. This is because it requires more space to store a public key than a hash of that key. Similarly, a *Multi-Signature* transaction is one that requires multiple signatures in order for the transaction to be spent. For example, requiring 2 out of 3 signatures in order to release the funds. These types of transactions support M of N schemes, for example, to provide redundancy and additional security features such as forms of escrow.

The *Pay-To-Script-Hash* transaction type was developed to allow a payment to be sent to the hash of a script. Although this construct seems strange at first, the motivation behind it was to allow the complexity of implementing scripts to be moved away from the party creating the transaction to the one wanting to spend it. For example, imagine that a company implements some type of internal security features to prevent its employees from running away with its money. These security features are implemented as a complex script that they require all customers to attach to their transactions. However, as a customer, I don't want to worry about creating these complex *locking* transaction scripts every time I wish to purchase something from the company. Instead, by using a *Pay-To-Script-Hash* transaction type the company can be responsible for implementing those scripts, and the customers can pay to the *bitcoin addresses* provided by the company, hiding these complexities.

Finally, the *Data-Output* transaction type, sometimes called the *OP_RETURN* transaction type, was developed to create *provably unspendable* transactions. Using this transaction type you can generate unspendable outputs that contain up to 40* bytes of arbitrary data. This data can be used to store any type of additional information associated with a transaction. The *OP_RETURN* transaction type was only later introduced as standard as a response to users wanting to store *Bitcoin* related data with various transactions. As mentioned in the *Blockchain* section above, one of the great properties of the *Bitcoin* blockchain is the ability to protect data in the chain from corruption, through *proof of work*. This has some interesting consequences and as a result, people have come up with ingenious and novel ways of using this transaction type to create a whole host of unique and unrelated applications. Examples include smart contracts, proof of existence and certification. As you increase the number of blocks on the chain, you increase the amount of work required to undo the blocks. This effectively adds more and more trust to the existence of the piece of information stored in a transaction. What's more, because every block is timestamped, it is possible to certify exactly when that data piece first existed.

*Note that as of *early 2015*, the *OP_RETURN* transaction type has had it's limitation of 40 bytes increased to 80 bytes [39].

As these novel applications have gained popularity among users, concerns have been raised about the amount of unnecessary information that the blockchain is currently storing. It has been argued that adding data to the blockchain that is unrelated to *Bitcoin* bloats the chain, making it bigger for no reason. Because the entire blockchain has to be replicated at every node in the network you effectively *waste* large amounts of space of many different nodes. However, decentralized replication and proof of work actually make the blockchain perfect for embedding information without worry of loss or corruption, and as such users have argued that these applications will only ever promote the use of *Bitcoin* as a currency.

Before the existence of the *OP_RETURN* transaction, users came up with other ways of embedding data into the blockchain. One of which was to take some piece of information, hash it to a fixed length and use the hash as the destination address of a new transaction. This had two implications. The first of which was that, because the destination address of the transaction does not correspond to any real *bitcoin address*, these transactions can never be spent, meaning that the amount of bitcoin provided as inputs to the transaction are forever lost, decreasing the total amount of bitcoin available globally. The second implication was that, because mining nodes store unspent transactions in main memory in order to allow efficient generation of new blocks, these types of transactions will too be stored in memory. However, because they can never be spent, they will exist forever in main-memory, effectively requiring miners to continuously increase the size of their RAM in order to continue operating. The motivation behind the *OP_RETURN* transaction was to prevent problems like this from occurring. By making the transaction *provably unspendable* through the transaction script, the miners will be able to identify these transactions and will not store them in main-memory. The embedded data however will still exist forever in the block chain.

2.1.6 Wallets

A *Bitcoin wallet* is a storage container that stores the access information for *Bitcoin accounts*. As previously mentioned, a *Bitcoin account* is a *private-public key-pair* and in order for someone to spend the funds in an account, they must have the correct *private key* to unlock those funds. All a wallet therefore needs to do is store a set of *private keys* as *public keys* are easily derivable from a *private keys*. A question one might naturally ask is why then would someone want more than 1 account, or 1 *private-public key-pair*? Typically the reason for this is anonymity. If someone were to only use one *public key* for all of their transactions, although you wouldn't know who they were, you would still be able to track all of their expenditures and incomes, and in doing so you might be able to infer something about them. To protect against this it is common practice for people to use a new account or *key-pair* for every transaction that they perform. This makes tracking payments and the flow of money much more difficult.

Although this might seem wasteful, as you effectively create a new account for every transaction, the length of a private key is 256 bits long. This means that there are 2^{256} different bitcoin accounts. This number is equivalent to around 10^{77} and is so sufficiently large that running out of addresses should not be a concern. Furthermore, it also means that the chance of any two people generating the same *private-public key-pair* is nearly impossible, assuming that the generation of key pairs is randomly and evenly distributed. Note that this may not be the case for incorrect or insecure implementations of account generators.

A *Bitcoin wallet* is therefore simply a *private key* store. It can be in the form of a hard drive, a CD, an application on your mobile phone or even a piece of paper that you keep in a safe. Anything that can store data can act as a *Bitcoin wallet.* Arguably, the biggest problem with *wallets* today is that if you lose your wallet, or the keys are removed, you permanently lose access to the money in those accounts. In fact over the past couple years there have been several unfortunate cases where people have lost their private keys and hence forfeited large amounts of bitcoin. In one case a man threw out an old computer hard drive on which he had stored his private keys, resulting in a loss of 7500 *BTC*, with an estimated value of £4.6 million [9] at the time. Likewise, because keys are regularly being generated for new transactions, wallets also need to be backed up on a regular basis to prevent data loss.

To protect against these problems there have been several developments in wallet technology over the past few years. These technologies range from software based wallets, such as *seeded hierarchical deterministic* wallets, to hardware based wallets, such as *Bitcoin Trezor* [113]. Each type of wallet has its own unique advantages and design purpose. Software based *seeded* wallets for instance, only need to be backed up once because the generation of new keys is deterministic upon some seed. Hardware based wallets on the other hand are designed to protect from viruses, cyber-attacks and malware. For example, figure 6 shows *Bitcoin Trezor*, a type of hardware based wallet. This wallet is designed to prevent *private key* theft. It does so by never exposing the *private keys* stored in the wallet when transactions are signed. This means that regardless of how insecure the computer is that is being used to generate the transactions, the *private keys* will not be made vulnerable. This type of offline *Bitcoin* storage is often referred to as *cold* storage.

Figure 6: Bitcoin Trezor Hardware based Wallet [113]

2.1.7 Alt Coins and Chains

A year after the initial *Bitcoin whitepaper* was released, a *reference implementation* [18] was published online by *Nakamoto.* This implementation was released under an open-source license, allowing the bitcoin community to verify its correctness and contribute to the project. Since then, the implementation has been revised and updated by many different bitcoin developers and enthusiasts worldwide. In 2011 *Nakamoto* withdrew from the community, leaving the primary control of development to a group of community volunteers.

By open-sourcing the project, *Nakamoto* also gave individuals the ability to fork the reference implementation and create their own digital currency based on its design. This decision has resulted in the release of hundreds of different online alternative currencies. Each currency providing some unique spin or variant on the original *Bitcoin* implementation. These alternative currencies are often referred to as *altcoins* and usually differ from *Bitcoin* through some small alterations. For example some modify the total number of coins that will be released over the currencies lifetime while others modify the speed at which blocks can be generated or the *proof of work* algorithm used to generate those blocks. Although many of these coins are based on bitcoin, some do not re-use any of its code and instead borrow the principles and ideas behind the currency. Nonetheless it is still common for these currencies to be called *altcoins*.

For example, one of the very first *altcoins*, *Litecoin* [73], was released in 2011.*Litecoin* is the second most successful digital currency, second only to *Bitcoin* itself. The primary differences between *Bitcoin* and *Litecoin* are that *Litecoin* has an average block mining time of 2.5 minutes, a total number of coins limited to 84 million and it uses a different *proof of work* algorithm called *Scrypt*. The change in the *proof of work* algorithm used by *Litecoin* means that the network is more resistant to rapidly accelerating *hardware* as the algorithm is very *memory intensive* and much more serialised than *SHA256* used by *Bitcoin*. This means that individuals don't require large investments in order to remain competitive when mining. They are not quickly outdone by specialised harware rigs, such as *application specific integrated circuits*, or *ASICs*. Furthermore, the increase in the speed of block mining means that the network can cope with transactions much more quickly, providing confirmations much earlier.

Another intersting example of an *altcoin* is *Curecoin* [43] that was released in 2013. *Curecoin* provides an interesting innovation on the foundation already set by *Bitcoin*. Rather than use *SHA256* as the *proof of work*, *Curecoin* uses a *protein folding* algorithm developed by *Stanford University*, called *Folding@Home* [115]. *Folding@Home* is an algorithm that simulates protein folding and performs scientific research calculations, the results of which are useful in helping to cure diseases such as Cancer and Alzheimer's. The idea is to replace the wasted computation in the *Bitcoin* network with computation that would help medical research at the same time. *Curecoin* has a current block generation time of 10 minutes, and an unlimited currency generation.

As well as innovation in *proof of work* mechanisms, there have also been several alternate currencies based on improving the anonymity of *Bitcoin* transactions. One such currency is *Bytecoin* [31], a currency launched in 2012 based on the *CryptoNote* [42] reference implementation. The idea behind *CryptoNote* and *Bytecoin* was to use *ring signatures*, a mechanism where transactions between parties are signed by multiple individuals. The idea is that the verifier of the transaction cannot distinguish the direct paticipants from the rest of the signing group. This makes blockchain analysis and transaction tracking much more difficult and thus increases the anonymity of the network.

Figure 7 shows the different logos for each of the digital currencies mentioned above. These logos are for *Bitcoin*, *Litecoin*, *Curecoin* and *Bytecoin* respectively.

(a) Bitcoin Logo [123] (b) Litecoin Logo [73] (c) Curecoin Logo [43] (d) Bytecoin Logo [31]

Figure 7: Logos of Various Digital Currencies

In addition to *altcoin* technology, the open-source nature of *Bitcoin* has also driven development in *altchain* technologies. These type of developments innovate on the block chaining algorithm used inside *Bitcoin* in order to achieve consensus on a variety of different problems. For instance, providing a decentralized *DNS registrar*, resource distribution and contract generation.

One concrete example of an *altchain* is *Bitmessage* [22]. *Bitmessage* is a distributed secure messaging service. It uses a block chaining algorithm to provide peer-to-peer and trust-less communication where the parties involved are kept anonymous and their messages hidden through encryption. *Bitmessage* circumvents the single point of failure associated with email servers, making denial of service attacks, eavesdropping and message observation much more difficult. Furthermore messages are not persistent, they only live for a certain period of time before they disappear from the chain.

2.1.8 Bitcoin 2.0

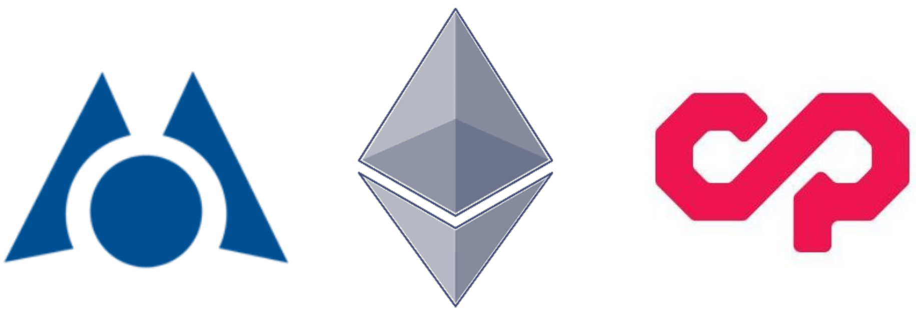
Just like alternative currencies and chains *Bitcoin* has also inspired the development of many different *metacoins* and *metachains*. These innovations build software layers and protocols directly on top of the *Bitcoin* blockchain, allowing them to support many different applications such as currencies inside currencies, or the creation of much more powerful protocols. The majority of these innovations use the `OP_RETURN` construct in *Bitcoin's* transaction scripts to embed pieces of metadata into the blockchain. These types of innovation are much more widely referred to as *Bitcoin 2.0*, effectively representing the next generation of *Bitcoin* technology.

One such example of a *metacoin* is *Mastercoin* [76]. *Mastercoin* is a protocol layer built on top of the bitcoin blockchain, similar to how *HTTP* uses *TCP/IP*. The idea of *Mastercoin* is to provide a framework and a set of tools that can be used to develop new applications. One example is supporting the creation of new *user currencies* where individuals can build their own customised currencies without needing to do any software development. *Mastercoin* uses a special *Bitcoin address*, called the *Exodus* address, to embed data into the blockchain. This allows it to differentiate normal transactions in the network from *Mastercoin* specific ones. Howevere there are plans for it to migrate to use the *OP_RETURN* transaction operator in the future. *Mastercoin* uses its own currency *MST* as a token for building *Mastercoin* transactions. The distribution of *MST* was based on early payments to the *Exodus* address, allowing users to effectively trade *Bitcoin* for *Mastercoin*.

Another example of a *Bitcoin 2.0* innovation is *Ethereum* [48]. *Ethereum* has been defined as a *"next generation smart contract and decentralized application platform"* [49], effectively acting as a *Turing-complete* contract programming language. *Ethereum* is not a copy of *Bitcoin*, but is it's own unique innovation, borrowing several principles from *Bitcoins* design. *Ethereum* models its blockchain as a state transition system and uses it to act as an abstraction layer for a *Turing-complete* programming language. This would allow anyone to write decentralized applications and smart contracts in a very simple way, often reducing the complexity of many existing *altchains* to a very few lines of *Ethereum* code. *Ethereum* uses an internal currrency called *Ether* to drive the protocol.

Finally, *Counterparty* [41] is another *Bitcoin* innovation that builds a protocol layer on top of a blockchain. *Counterparty* provides peer-to-peer financial tools and a platform on which to create smart contracts, perform asset exchange and generate custom tokens. *Counterparty* is actually a port of *Ethereums* open-source reference implementation. Instead of using a new custom block chain to build upon, *Counterparty* uses the *Bitcoin blockchain* as its foundation, arguing that there is no need to create a new blockchain as suggested by *Ethereum*. *Counterparty* uses the *OP_RETURN* script construct to embed data into the blockchain and its own internal currency *XCP* to drive the protocol. In contrast to how *Mastercoin* and *Ethereum* distributes their tokens, *Counterparty* uses a *proof-of-burn* [124] scheme where miners show that they have burnt an amount of *Bitcoin* in order to recieve *XCP*. Burning *Bitcoin* effectively means sending an amount of *Bitcoin* to an unspendable address. This act effectively bootstraps value, demonstrating a belief that by burning something valuable, you believe the purpose of that burn to have value too, i.e. you burn *Bitcoin* for some *XCP*. In doing this, *XCP* is given value through the lost *Bitcoin*.

Figure 8 shows the different logos for each of the three *Bitcoin 2.0* technologies mentioned above. These logos are for *Mastercoin*, *Ethereum* and *Counterparty* respectively.



(a) Mastercoin Logo [76] (b) Ethereum Logo [49] (c) Counterparty Logo [41]

Figure 8: Logos of Various *Bitcoin 2.0* Technologies

2.2 Online Betting and Gambling

2.2.1 Overview

In addition to virtual and digital currencies, the Internet has also given birth to the global online gambling and betting marketplace. With the rapid expansion of the World Wide Web, the worlds first online casino, *InterCasino*, was opened in 1996 [68]. It was the first casino to offer online betting and wagering in fiat currency, making use of the Internet as an open, wide-spread and instantly accessible medium. Online betting and gambling has since grown rapidly. So much so that, in 2012 it was estimated to be worth in excess of 2 billion pounds in the U.K alone [36].

Despite the border-less nature and freedom of the Internet, online betting is a very highly regulated and complex sector. Different countries around the world hold different laws regarding online betting. Unsurprisingly, each country also has its own unique definition to describe exactly what constitutes betting. For example, in the United States it is against federal law for websites to take *sports* bets over the Internet. They can however take bets from casinos or poker online, provided the state in which the participants reside don't have locally conflicting laws [1]. In South Korea however, any type of gambling or betting is illegal, regardless of whether or not it occurs online. This is true in all cases except if the person placing the bet is a tourist, not of Korean, Vietnamese or Nepalese citizenship. In the United Kingdom, gambling is legal both online and off, provided the participants are over 18 years of age. Unfortunately, things become even more complicated when gambling occurs online across several different jurisdictions. For example, an American citizen in Texas placing a bet on a sports website registered in the U.K, on a server running in France.

The outcome and resolution of bets are also highly complicated in their own respects. Gambling agencies post long and intricate terms of conditions surrounding the outcome of a bet and the potential situations that can arise. For example, what happens if a player retires in a game of tennis without having finished the match [2]. Or if an athlete wins an Olympic event, but subsequently has their medal revoked after failing a drugs test [111]. It is the details of these exceptional, but not uncommon, circumstances that make betting a fascinating and complex area.

In order to understand the various types of wagers and bets available, it useful to understand exactly what betting and gambling are. It has been said that gambling is the act of *"wagering money, or items, on an event with an uncertain outcome"* [128]. The primary motivation for gambling is to attain a greater wealth or value than what was originally staked. It is often agreed that any type of wager, regardless of its nature, contains 3 key elements, *"prize, consideration* and *chance"* [100]. The first element, *prize*, is the incentive or motivation for the parties involved. The *prize* provides the participants with a reason to stake something of theirs in order to gain something of even greater worth. The second element, *consideration*, requires a specific act by the participants, such as risk, in order to be considered for the reward or *prize.* The third element, *chance*, requires that the outcome of the event or wager is not predetermined or predefined. It may involve some element of skill or ability but it should not be deterministic, otherwise there would be no incentive for the parties to bet against each other.

Wagers can come in many different forms. As long as the 3 key elements are present, a wager can be placed on just about anything. Traditionally, *casino betting* often encompass wagers on the outcomes of various *games of chance.* The participants or *bettors* can compete directly against the *casino* (or *house*), or against each other. Examples of casino games include *craps, roulette, poker* and *blackjack.* Some of these games are considered pure games of *chance* and others a hybrid between *chance* and *skill. Roulette*, for example, is considered a pure game of chance [23]. A mechanical wheel is spun and a small ball is placed into the wheel. Eventually the wheel stops and the ball lands on a number and colour. This game, if implemented correctly, should be random and fair, with the probability of the ball landing on each of the numbers being evenly distributed. Because of this casinos often implement a *house edge*, where the amount paid out to any winning player is slightly less than would be expected based on probability. This moves the odds into the casinos favour so that over a long period of time the casino wins more money, on average, than it loses. In contrast to *roulette*, *poker*, is often considered a game of *skill.* This is because the outcome of the game not only depends on the *luck* of the cards dealt, but also the moves and strategies taken by the player.

Sports betting is a different type of betting. *Sports betting* typically involves wagers on the outcome of various sporting events, such as tennis matches or football games. Participants can wager, for example, that a specific team or individual might win a tournament, or that the points difference between any two teams in a game might be greater than some number. The types of bets that can be placed on any event are usually called the *betting markets*. Any specific event, such as the *2015 NFL Super Bowl Final*, will typically have in the range of 1 to 30 markets, each of which varies from `‘‘which team will win?’’`, to `‘‘the total number of yards run in the 3rd quarter’’`. *Sports betting* is extremely popular in the UK and has an estimated net worth of around *£650 million* [61]. Other popular types of betting outside of casinos and sports include *lottery* betting, such as taking part in the *UK National Lottery* [75], and buying *scratch cards*. *Scratch cards* are small cards that can be bought from select stores. They are scratched to reveal some combination of numbers and symbols, which if considered a winning combination can result in prize money or other valuable items. Figure 9 shows a *National Lottery* scratch card.

Figure 9: National Lottery Scratch Card [74]

2.2.2 Sportsbooks

A *sportsbook* is a service that accepts bets on various sporting events. *Sportsbooks* traditionally feature *fixed-odds* and *parimutuel* betting on different markets. *Fixed-odds* betting is a style of betting where specific *odds* are calculated for each market and offered to bettors. The *odds* typically correlate with the probability of the outcome and may include a *house edge*. A *sportsbook* is said to be *balanced* if the overall outcome is in favour of the individual offering the book, often called the *bookie* or *bookmaker*. This, for example, means that the *bookie* stands to make an overall profit regardless of the outcome of the various markets. Creating a *balanced* book is not straight forward as it is impossible to know the true probabilities of any event occurring. As such *odds* are typically calculated using various pieces of information and heuristics. It is worth noting here that an *arbitrage* situation is one that can arise when a bettor takes advantage of a price difference between two markets. This results in the bettor being guaranteed a profit, regardless of how the market is settled. *Arbitrage* situations typically arise due to different odds being offered by different *bookmakers*.

Fixed-odds are traditionally offered in three different formats, *fractional odds*, *decimal odds*, and *moneyline odds*. *Fractional odds* are most common in the UK and represent the total amount of money that will be paid to the bettor, should they win. For example, the odds of *10/1*, read as 10 to 1, mean that for every £1 staked, the bettor will receive £10 if they win. Therefore if the bettor bets £2 on some outcome at *10/1* odds, and that outcome is correct, they will receive £20, in addition to the originally staked £2, making £22 in total. *Decimal odds* on the other hand are a little simpler. They differ from *fractional odds* in that, the odds when multiplied to the original stake, represent the total amount of money won. For example, if you are given odds of *5.00*, that means for any amount you bet, multiple it by *5.00* and that is the total amount you will receive if you are correct. So, if you bet £1 at odds of *5.00*, you will receive £5 if you are correct, and if you bet £2 you will get £10. *Decimal odds* are simpler than *fractional odds* because you don’t need to add your original stake back onto the total amount of winnings. Decimal odds are used more commonly in Australia, New Zealand and Canada.

Moneyline odds are a little more complicated and are typically favoured by American *bookmakers. Moneyline odds* can be quoted as either being *positive* or *negative.* If the odds offered are *positive*, such as *+500*, this represents how much money will be won on a *$100* stake. In this case, the winner will receive *$500* plus their original stake of *$100*, making these odds equivalent to *5/1* in *fractional odds* format. If odds are *negative*, such as *-500*, this represents how much must be wagered to win *$100.* In this case, odds of *-500* mean that in order for the bettor to win *$100*, they must wager *$500*, making these odds equivalent to *1/5* in *fractional odds* format. Even odds, (eg. *1/1* in *fractional* format) can be represented as *+100* or *-100.*

Parimutuel betting is another style of betting typically offered by *bookmakers.* In contrast to *fixed-odds* betting, *parimutuel* betting is where all bets on a particular event are *pooled* together and when the event is over, the pool is shared among the winners. *Parimutuel* betting is quite common for events where participants finish in a ranked order, such as in horse or greyhound racing. For *parimutuel* betting the final payout or odds are not determined until the pool is closed and the *commission* removed. To illustrate this, in a horse race with 3 competitors, bettors place money on who they think would win. When the race begins, and the pool is closed, there is £100 on horse 1, £50 on horse 2 and £150 on horse 3. When the event is finished and horse 3 declared the winner, the payout would be calculated as follows. First, the *commission* would be removed from the pool, lets assume there is 0 commission for the sake of this example. Then the total amount in the pool, £300, would be split relatively between the parties who bet on horse 3. The odds for horse 3 in this example would therefore be calculated as $£300 \div £200 = 1.5$, so *decimal odds* of 1.5, and *fractional odds* of 1/2. Before an event begins and the pool for that event is closed *bookmakers* typically provide estimates for the current odds in the pool, should no more bets be placed. This allows bettors to have some idea of the current market return.

Although *fixed-odds* and *parimutuel* betting are the more common types of betting, many *bookmakers* offer various interesting twists and opportunities to their customers. For example, *in-play* betting is a betting type where bettors can place bets while an event is currently happening. *Second-half* betting is where bets can be placed only on the outcome of the second half of a game. And *Parlay* betting allows bettors to chain bets together to receive higher pay-outs and improved odds. Many of these betting styles depend on the type of event being played, the different possible outcomes and the *bookmaker.*

2.2.3 Betting Exchanges

In contrast to *sportsbooks*, *betting exchanges* are entities that support peer-to-peer betting. Rather than betting against a *bookie* or *bookmaker*, *betting exchanges* allow bettors to compete directly against one another. This typically means that the *odds* offered on *betting exchanges* are much more attractive than those offerred by *sportsbooks* as there is a lower cost overhead. In addition to this, *betting exchanges* also offer increased flexibility as bets can be placed both for and against a specific outcome. For example, in a horse race, if you think a specific horse will win, you *back* them. This is a bet placed for a specific horse to win. However if you think that a horse will lose, you *lay* a bet against them. This means that in a race with many different horses, for your bet to win, any horse can win except for the one you chose to lay. This allows increased flexibility when matching opposing bettors in the exchange, even if some people find the idea of betting that someone will lose dubious.

Typically, the operator of the *betting exchange* takes a small *commission* on the winning bets, and takes no commission on losing bets. The amount taken is usually small enough that the odds are still much more attractive than those offered by *sportsbooks.* In addition to this, *betting exchanges* also offer less restrictions when it comes to betting. *Sportsbooks* for example may limit the activities of a bettor, such as the frequency or total amount wagered. This is common if someone wins too much money, putting the *bookmaker* at a loss. This is not the case with *betting exchanges.* As long as there is someone to match the bets placed, the size and frequency of those bets are unrestricted. *Betting exchanges* therefore tend to thrive when there is high market liquidity, as this drives up the number of bets matched. Although *betting exchanges* are able to offer better odds, they still lack many of the exciting features offered by *sportsbooks*, such as *parlay* betting.

At this point, it is worth understanding how *back* bets and *lay* bets are matched against one another in a single market. As previously mentioned, a market for an event can have several outcomes. For example, in a horse race with 8 horses, the market for *"Who will win the race?"* can have 8 possible outcomes, one for each horse (or *runner*). This means that a betting exchange can match bets in a single market using several possible ways. One way is to match bets across a single outcome. This means one user chooses to *back* a specific outcome, and the other user chooses to *lay* that very same outcome. If the odds proposed by the users are appropriate, the bets can be matched against one another.

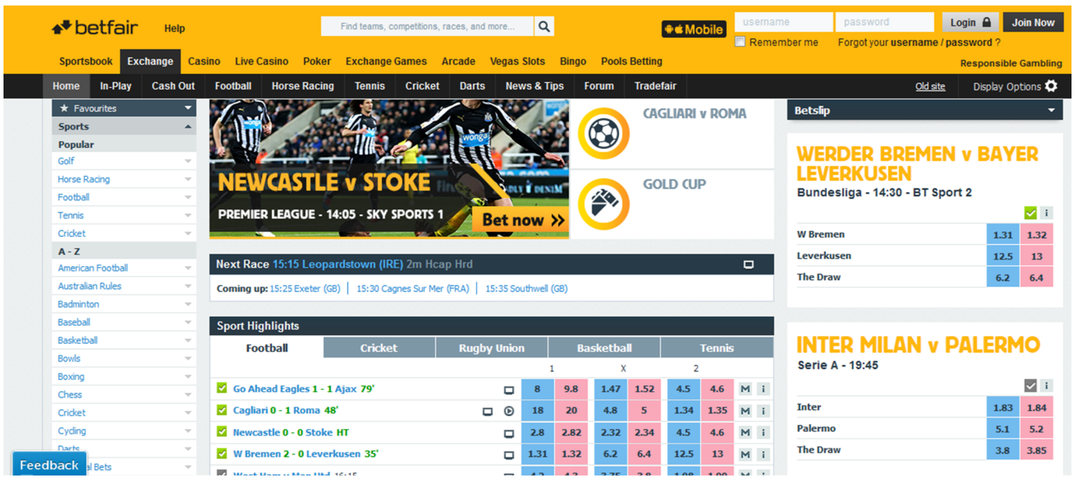
Another possible way to match bets is when the market has only 2 possible outcomes, for example if there are only 2 horses in a single race. Because there can only be 2 outcomes (either the first horse wins, or the second one does) bets can be matched in a slightly more flexible manner. Matches across a single outcome can still happen as before, but now, we can also match *back-back* bets on the two runners, and *lay-lay* bets on the two runners. For example, if someone chooses to *back* horse 1, and someone else chooses to *back* horse 2, because there can only be 2 possible outcomes, these bets can be matched against each other (e.g. the first person said horse 1 was going to win, but the second person said horse 2 was going to win). The same goes for a *lay* on horse 1, and a *lay* on horse 2. Note that we are implicitly making the assumption that we do not worry about an exceptional outcome, one where no horse wins.

In order for bets to be deemed appropriate for a match, we need to consider the odds that each bet has been proposed at. For the sake of example, let's assume that we are dealing with *decimal odds*. As mentioned above, *decimal odds* are odds in decimal format (e.g. 1.5, or 2.3, or 3.7). If you place a bet at decimal odds of y, for an amount z, and your bet wins, the total amount you get back is $y \times z$. For example, if you placed a bet of £5 at odds of 1.5, and your bet won, you will receive *£*5 × 1.5, 7.50. Naturally, it doesn't make sense to place a bet with odds less than or equal to 1, because even if your bet wins, you would still end up with less than or the same amount of money.

In order for bets to be deemed appropriate for matching the *back* odds and *lay* odds need to correlate. For example, if someone lays a *back* bet of £5 at odds of *1.5*, then that person should win £7.50 in total if their bet is a winning bet. This means that a *lay* bet wanting to match this, needs to put £2.50 into the pot. Now, because the total pot would be £7.50, the odds at which they are laying is *3.0* (i.e. they put £2.50 in, and if they win, will get £7.50 out). So in order for bets to matched, the odds at which they are proposed need to have this relationship. The same goes for *back-back* matches and *lay-lay* matches in markets with only 2 outcomes.

Note that things can get slightly more complicated when we see that bet matches do not have to be *complete*, but that *partial* matches are allowed. So in the example above, the *lay* bettor could choose to *completely* match the *back* bet by placing £2.50 into the pot at *lay* odds of 3.0, but they could also *partially* match the *back* bet by only placing £1.25 into the pot at odds of 3.0. This means that the *back* bet would only be half matched, and so someone else could decide to come in and *lay* another £1.25 to match the remaining £2.50 of the *back* bet.

Examples of two very large and popular online *betting exchanges* are *Betfair.com* [13] and *Bet-Daq.com* [12]. One thing notable about *betting exchanges* is that the identities of the bettors who's bets are matched are kept anonymous. This means that if you a place a bet and it is matched by someone else, only the exchange knows how that bet was matched and which parties were involved. Furthermore, because online *betting exchanges* only require a minimal amount of personal informational, and do not check if that information is valid, the level of anonymity is much higher than that of a *sportsbook*. Figure 10 shows a screen shot of the *Betfair betting exchange* website.



Next Race 15:15 Leopardstown (IRE) 2m Hcap Hrd

Coming up: 15:25 Exeter (GB) | 15:30 Cagnes Sur Mer (FRA) | 15:35 Southwell (GB)

Sport Highlights

Football | Cricket | Rugby Union | Basketball | Tennis

	1		X		2	
Go Ahead Eagles 1 - 1 Ajax 79'	8	9.8	1.47	1.52	4.5	4.6
Cagliari 0 - 1 Roma 48'	18	20	4.8	5	1.34	1.35
Newcastle 0 - 0 Stoke HT	2.8	2.82	2.32	2.34	4.5	4.6
W Bremen 2 - 0 Leverkusen 35'	1.31	1.32	6.2	6.4	12.5	13

WERDER BREMEN v BAYER LEVERKUSEN

Bundesliga - 14:30 - BT Sport 2

W Bremen	1.31	1.32
Leverkusen	12.5	13
The Draw	6.2	6.4

INTER MILAN v PALERMO

Serie A - 19:45

Inter	1.83	1.84
Palermo	5.1	5.2
The Draw	3.8	3.85

Figure 10: Betfair Online Betting Exchange [13]

2.2.4 Bet Resolution and Settlement

When placing a bet or wager via a *sportsbook*, or *betting exchange*, there are very long and intricate *terms and conditions* surrounding the outcome of a bet. *Betfair*, for example, provides a lengthy set of *rules and regulations* [14] outlining the ways in which way a bet can be settled under various circumstances. These regulations include a set of *general rules* as well as *sports specific rules*. For example, *Betfair* provides a sports specific clause relating to *shirt number* bets in *football*:

```
- 'Shirt numbers' bets will refer to the shirt number allocated at the start
  of the match.
- 'Shirt numbers' bets will include own-goal scorers.
- Any player whose shirt bears no number will be allocated the number 12.
```

Similarly *Betfair* also includes a clause relating to the outcome of tennis matches where a player retires or is disqualified:

```
- If a player or pairing retires or is disqualified in any match, the player
  or pairing progressing to the next round (or winning the tournament in
  the case of a final) will be deemed the winner.
- However if less than one set has been completed at the time of the
  retirement or disqualification then all bets relating to that individual
  match will be void.
```

These terms and conditions have to be considered carefully by bettors, especially *arbitrage* bettors, as the appearance of an arbitrage situation may not be so clear cut when the markets being played off against each other contain slightly different terms and conditions. Likewise, the *sportsbooks* and *betting exchanges* that offer these bets need to make sure to cover all the possible cases and situations that can arise surrounding a bet, otherwise they may be left exposed to unexpected outcomes.

As a result of this complexity, it is not uncommon for disputes to be raised regarding the settlement of a bet. As such, UK *bookmakers* and *betting exchanges* are required to register with third party adjudication services in order to deal with these disputes. *Betfair* for example is registered with the *Independent Betting Adjudication Service*, or *IBAS* [63]. *IBAS* is a third party organisation that acts an impartial adjudicator for disputes that arise between bet makers and bet operators in the United Kingdom. It was reported to have awarded over £365,000 to customers over disputes in 2007 [62]. Third party adjudication services typically only ever get involved with disputes when the bet maker has gone through the internal dispute process of the bet provider, but has been unhappy with the result.

In addition to third party adjudication services, there is also a regulatory body in the UK responsible for betting and gambling. This body is called the *UK Gambling Commission* [57] and was set up in 2007 as a direct result of the *2005 Gambling Act* [56]. The purpose of the *Gambling Commission* is *"to keep crime out of gambling, to ensure that gambling is conducted fairly and openly, and to protect children and vulnerable people."* It is therefore responsible for issuing licenses to gambling operators, prosecuting illegal gambling and for advising government on gambling related problems.

The outcome of various sports events and matches are typically declared by the official authorities of that sport. For example the *Association of Tennis Professionals*, or *ATP*, are considered the official authority on tennis rankings and match results worldwide. Likewise, the *National Football League*, or *NFL* are considered the official authorities on American football games and results. *Betting exchanges* and *sportsbooks* will usually settle bets using the result announced by these sports authorities and the terms and conditions surrounding that bet.

2.3 Related Work

There have been many developments in the *Bitcoin betting exchange* and *proof based application* markets over the past few years. These include the invention of various *Bitcoin* specific *casino* games as well as *Bitcoin oracles.* This section aims to outline, analyse and compare these bodies of work, partitioning the various solutions into groups based on their purpose and design choices. We pay particular attention to the attributes that make these solutions unique and focus on the comparison between them and our project. These groups include *fiat-backed* betting exchanges, *Bitcoin* sportsbooks and betting exchanges, *Bitcoin* based casino games, proof based data applications, *Bitcoin oracles* and *Bitcoin 2.0* betting systems.

2.3.1 DirectBet

DirectBet[46] is a *Bitcoin* based *sportsbook.* It offers fixed-odds betting on various sports events, such as Tennis, American Football and Basketball, as well as several casino games, such as *All in Poker. DirectBet* is a *fiat-backed* sportsbook, meaning that it is internally backed by a fiat based betting exchange, in this case *Betfair*[13]. Whenever a bet is placed on *DirectBet*, a bet for the equivalent amount of *fiat* currency is placed on the same outcome in the *Betfair* market. This means that if the bettor wins, they receive their winnings in *Bitcoin* from *DirectBet*, and *DirectBet* receives its winnings from *Betfair* in *fiat* currency. If the bettor loses, *DirectBet* keeps the staked *Bitcoin* but forfeits the *fiat* currency it paid to *Betfair.*

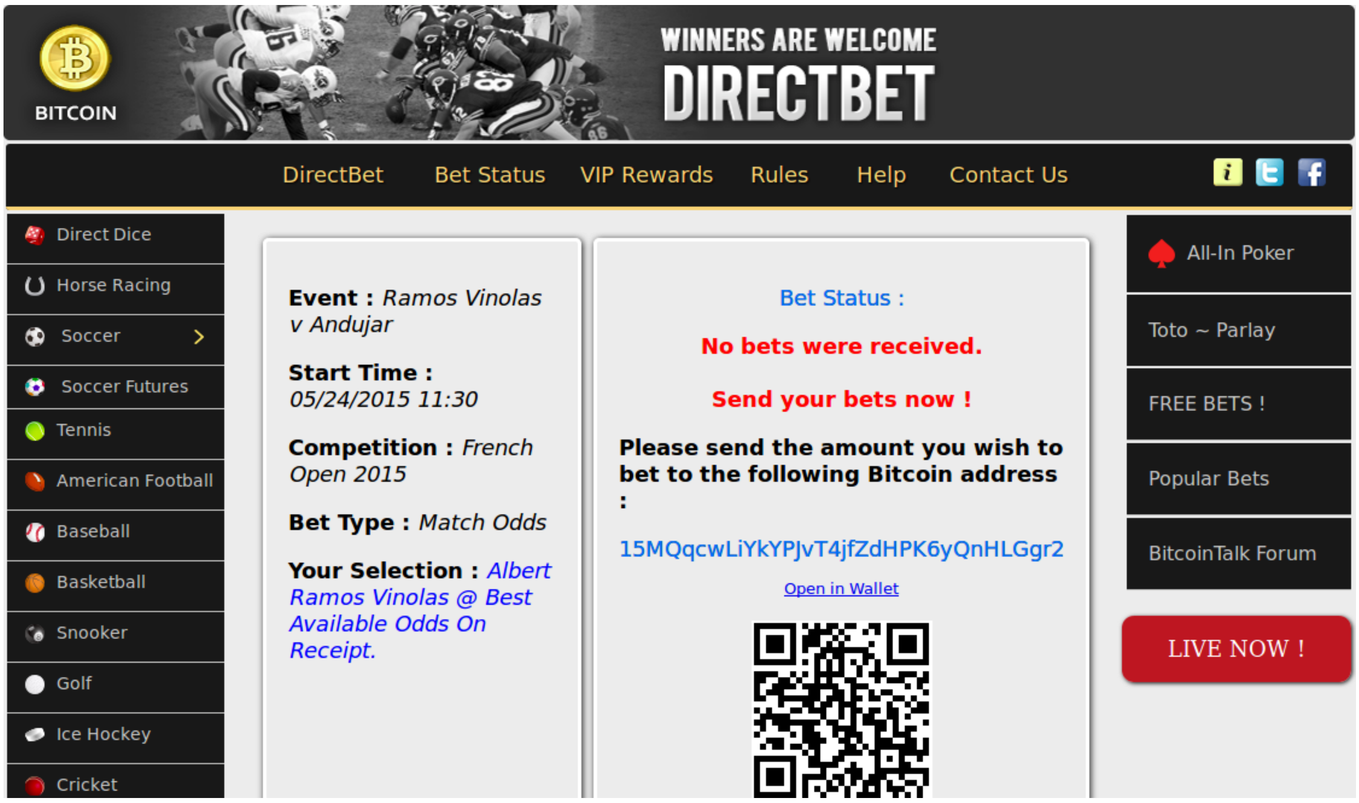
In order to remain profitable *DirectBet* adds a house-edge to the odds offered by the *Betfair* betting exchange. This means that regardless of the outcome of the various bets they will still make a profit. This will either be in the form of *Bitcoin* or in *fiat* currency. *DirectBet* does not require participants to register or create an account when betting. Instead, a unique *Bitcoin* address is generated for each event, outcome, odd selection and session. The bettor, if interested in the bet will create a transaction sending their *Bitcoins* to that specific address. This bet will then be registered with *DirectBet* and upon resolution all winnings will be transferred to the *Bitcoin* address entered when the bet session was created.

Unlike many other sportsbooks and betting exchanges, *DirectBet* does not ask users to deposit funds into an account before placing a bet. This is in contrast to the majority of online betting exchanges, such as *Betfair* and *BetDaq.* Instead, payments only occur when the bet is placed and there is no withdrawal procedure, any winnings are paid directly to the user immediately after the bettle has been settled. This means that the amount of time a user needs to trust *DirectBet* with their money is dramatically reduced, a point that *DirectBet* argues is unique to them. Furthermore, by not forcing users to create accounts, it maintains the anonymity of the participants involved. This is also another property unique to them. *DirectBet* does however offer bettors the option of associating an email address with each bet. This provides them with a bet confirmation and customer loyalty rewards. In addition to supporting *Bitcoin*, *DirectBet* also accept payments in *Dogecoin, Litecoin* and *Darkcoin*, allowing a greater number of users to use their service.

Although *DirectBet* seems to be quite popular, with multiple bets being placed each day [45], there are several disadvantages to their solution. The first is that they present no way to prove that a bet has been placed. This means that if a winning bet is placed on *DirectBet* and the winnings are not paid to the bettor, the bettor has no conclusive way to dispute their bet, meaning it is the bettors word against *DirectBet.* Although bettors can choose to receive an optional confirmation email, this is not a sound solution as emails can be easily spoofed or falsified. Furthermore, the initial transaction created by the bettor sending their bitcoins to *DirectBet* cannot be used as proof, because you cannot argue that the bitcoin address to which you sent your money (for the bet) belonged to *DirectBet* in the first place. Even if you provided a link to your bet page, there is nothing stopping *DirectBet* from deleting that page and claiming the bet never existed.

This means that if *DirectBet* were to act dishonestly and remove all traces of a bet from the user facing side of their website, you would have no conclusive evidence to present against them. This is actually one of the biggest problems with *Bitcoin* betting exchanges today, resulting in the creation of many dishonest betting exchanges, betting scams and hence disputes.

Another problem with the *DirectBet* model is that they rely entirely on the operation of *Betfair*. Whenever the *Betfair* exchange is down or not working, *DirectBet* is unable to function too. This is not ideal in the case that some bets might still be matchable across the *Bitcoin* network without the need for *Betfair*, and such it effectively costs them money. Finally, the large house edge that *DirectBet* add to some of their bets often detracts users from using their service. The size of this house edge is more a consequence of profit for *DirectBet* than it is for the cut that *Betfair* take on winning bets. Figure 11 shows a screenshot of an unpaid bet on the *DirectBet* website.



Event : *Ramos Vinolas v Andujar*

Start Time : *05/24/2015 11:30*

Competition : *French Open 2015*

Bet Type : *Match Odds*

Your Selection : *Albert Ramos Vinolas @ Best Available Odds On Receipt.*

Bet Status :

No bets were received.

Send your bets now !

Please send the amount you wish to bet to the following Bitcoin address :

15MQqcwLiYkYPJvT4jfZdHPK6yQnHLGgr2

Figure 11: DirectBet Website [46]

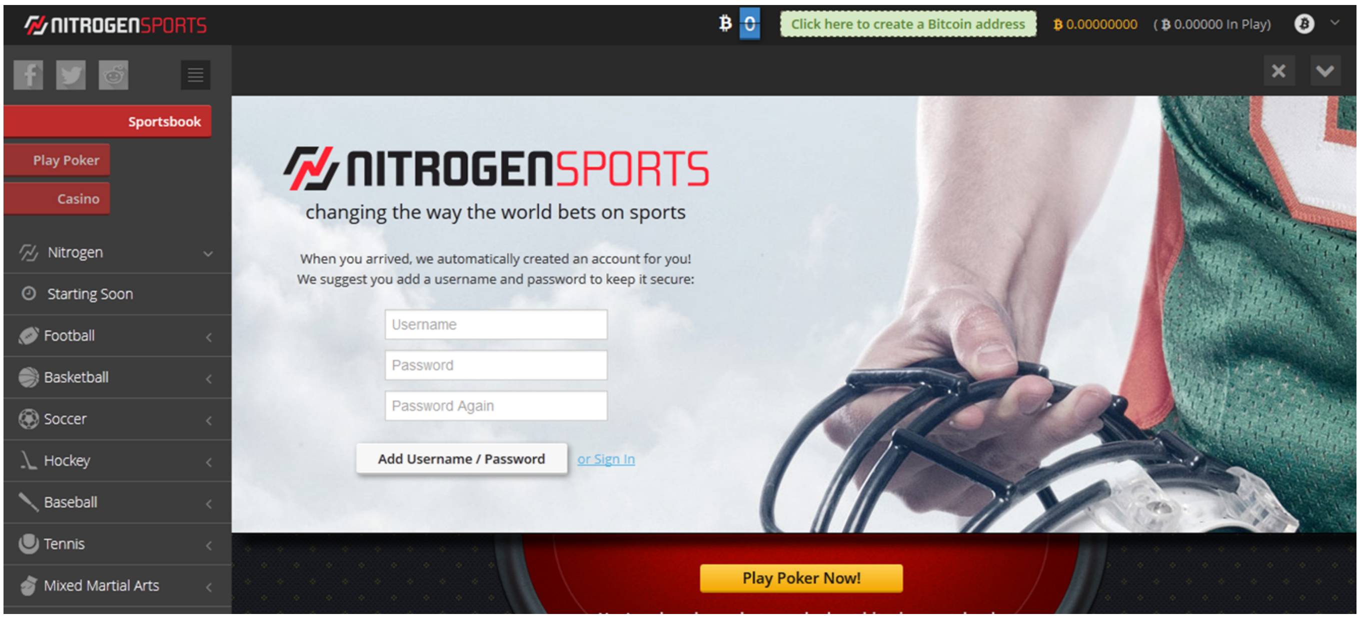
2.3.2 NitrogenSports & Anonibet

NitrogenSports and *Anonibet* are two examples of generic *Bitcoin* sportsbooks. What makes them different from *DirectBet*, analysed above, is that they are not backed by any *fiat-based* currency betting exchanges. These sportsbooks instead operate only in *Bitcoin* and the odds offered are calculated directly by the bookmakers themselves. In contrast to *DirectBet*, these sportsbooks require users to create accounts and to deposit funds into those accounts before they are able to bet. This requires users to trust the sportsbooks with their money for longer periods of time, and also negatively impacts their anonymity. This is because the sportsbooks can now associate bitcoin public keys with their user accounts, effectively allowing them to gather information on which bitcoin accounts paid money into which sportsbook accounts, and thus which sets of bitcoin public keys are likely to be owned by the same person.

Figure 12 shows a screenshot of the *NitrogenSports* website. *NitrogenSports* offers bets on various sports events, such as American Football, Basketball and Soccer, as well several casino games, such as Poker and Blackjack. In addition to these, it also supports parlay betting and live in play betting. One interesting feature to note about *NitrogenSports* is that it automatically creates new accounts for individuals based upon their browser cookies. These accounts can be linked to email addresses and usernames and are represented by long character strings. By default you only need to be embed the string into the website url to access the account. One example of an account url is:

```
https://nitrogensports.eu/u/d4f6515f888ab17cef3e245f10d43e7dca42e72b2ef98cf5dc5e6e407e5e6744
```

Although the chance of correctly guessing an account id is fairly low given the length of the string, it does pose a potential security problem, and as such *NitrogenSports* offers the ability to set a password and provides support for two-factor authentication. These features are not however used by default.



When you arrived, we automatically created an account for you!
We suggest you add a username and password to keep it secure:

Figure 12: NitrogenSports Website [84]

Figure 13 shows a screenshot of the *AnoniBet* website. *AnoniBet* is almost identical in functionality to *NitrogenSports*, except that it offers bets on a different set of sports and casino games. It also provides support for mobile and requires users to create an explicit account using a valid email address.

A property present in both *NitrogrenSports* and *AnoniBet*, along with *DirectBet*, is that they provide no conclusive way to prove that a bet was made. Users must again trust the sportsbooks to operate honestly and in the case of disputes or dishonest operation have no verifiable or conclusive evidence to use against them. This opens up the doors to scams and discourages users from using their services.

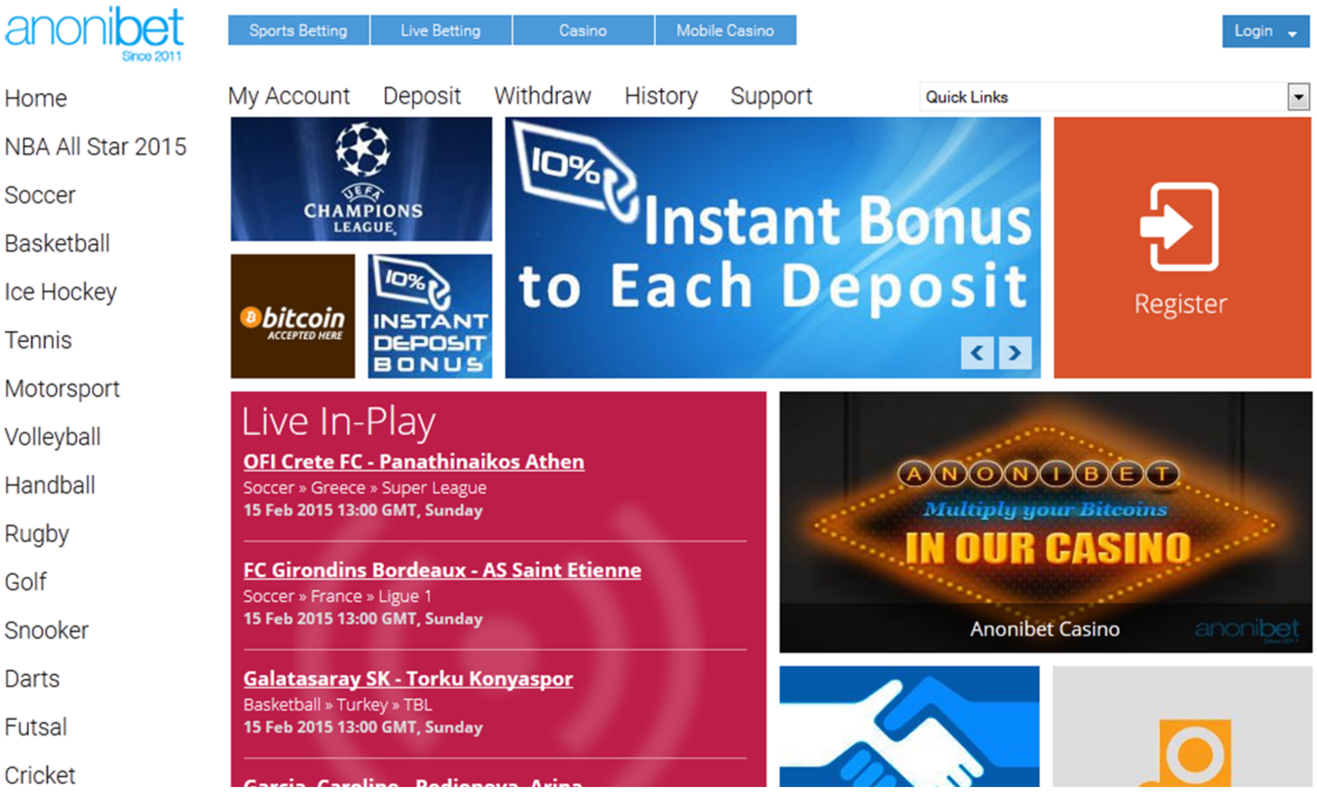


Figure 13: AnoniBet Website [5]

2.3.3 Betmoose, BitBet & BetBTC

Betmoose [15], *BitBet* [16] and *BetBTC* [11] are three examples of *Bitcoin* betting exchanges. In contrast to *Bitcoin* sportsbooks the bets are matched with other users and not the betting operators or an external exchange. *Betmoose* and *BitBet* offer parimutual style betting, where bets on a single event or outcome are pooled together and shared among the participants. The flexibility in this type of betting style means that these sites offer bets on a wide range of different topics, such as *"Who will win Best Picture in the 87th Academy Awards?"* or *"Will the price of 1 ounce of gold fall below $1175 USD on March 1st, 2015?"*. *BetBTC* on the other hand offers fixed-odds betting on various sports events, such as Hockey and Soccer. *BetBTC* is closer to a typical fixed-odds betting exchange as it allows both *back* and *lay* bets to be placed on the various events. *Betmoose* and *BetBTC* require users to create accounts before placing bets and *BitBet* requires no such registration. One thing that all three betting exchanges have in common is that they do not require users to deposit funds into an account before placing a bet. This, as discussed earlier, is better from the user perspective.

What's interesting to note about these betting exchanges is the way in which bets are resolved. *Betmoose*, for instance, offers parimutuel style betting where users, called hosts, create the topics and then are responsible for resolving the outcome of those topics. *Betmoose* allows users 24 hours to dispute the resolutions made by the hosts of those topics. If a dispute is raised, or the host fails to resolve a bet, the host loses their commission and a *Betmoose* moderator manually reviews and resolves the bet. It is therefore in the hosts best interest to resolve the bet correctly. It's important to note here that once the bet has been resolved manually by a *Betmoose* moderator, the participants have no way to challenge or dispute that decision.

Betmoose also suffers from the same problems as *DirectBet*, *NitrogenSports* and *AnoniBet* in that there is no way for users to verify the honest operation of those services. Instead users are forced simply to trust them to operate correctly. Although *BitBet* does not allow hosts to resolve bets and instead uses moderators to resolve them manually, this process also suffers the same problem. *BetBTC*, although not offering parimutuel style betting, face the same issues with sports betting. Figure 14 shows the logos of *BetMoose*, *BitBet* and *BetBTC* respectively.

(a) Betmoose Logo [15] (b) BitBet Logo [16] (c) BetBTC Logo [11]

Figure 14: Logos of Various Bitcoin Betting Exchanges

2.3.4 Satoshi Dice & Satoshi Bet

Satoshi Dice [44] and *Satoshi Bet* [10] are two examples of websites that offer *Bitcoin* betting on *provable-fair* casino games. *Provably-fair* casino games are games in which the participants are able to verify each round of execution for fair and honest gameplay. This allows players to check that the betting website implements each game correctly, providing fair chances of winning to its users. *Satoshi Dice*, for example, provides a *Blockchain-based Bitcoin Dice* game that is provable fair. These games work similarly to a normal betting website. Users sign up for an account, deposit an amount of bitcoin into their account and place a bet on the outcome of some event. In the case of *Satoshi Dice* the event being bet on is the outcome of a roll of dice. This dice has **65536** possible outcomes, each a number from **0** to **65535**. Users can then select exactly what condition they wish to bet on, for instance, they might bet that the dice roll is less than a specific number. For the sake of example lets say this number is 32768. If implemented fairly, this should have a 50 percent chance of happening on a completely random roll of the die. Having selected their bet, users can then see the roll outcome and instantly be paid any winnings.

What makes this game *provably-fair* is the way in which the outcome of the dice roll is generated. *Satoshi Dice* implement this as follows. First, they create a system *secret* at the start of each day that is known only by them. This *secret* is hashed and the hash published online. The outcome roll number that is then generated on a dice roll is calculated by hashing a *client seed* (a random number provided by the bettor), part of their bitcoin transaction in the blockchain, and part of *Satoshi Dice's secret*. 4 hexadecimal digits are then selected from the resulting hash and this is used to represent the final outcome of the dice roll, determining if the bet was won or lost. The way these digits are chosen from the hash depends directly on the *secret*.

In order for users to verify the outcomes of their bets, the *secrets* used by *Satoshi Dice* are published online at the end of every day. Users can test the various properties of these *secrets* and see that everything hashes to what it should, verifying the derivation of their result. In order to prevent bettors from gaming the system and carefully constructing bets that have higher chances of winning, the *secrets* are only revealed the following day, when no longer in use. *Satoshi Dice* add a house edge of 1.9% to every bet placed online in order to remain profitable. Likewise, *Satoshi Bet* also provide various provably-fair casino games. These include Roulette, Blackjack and Poker, and each of them has their own unique methods of verification and corresponding house edge. Figure 15 shows the logos of *Satoshi Dice* and *Satoshi Bet*.

satoshiDICE

(a) Satoshi Dice Logo [44]

SatoshiBet

(b) Satoshi Bet Logo [10]

Figure 15: Logos of Satoshi Dice and Satoshi Bet

One obvious disadvantage with *Satoshi Dice* and *Satoshi Bet* is that users are required to create accounts and deposit funds before they can place bets. As discussed above, this negatively impacts the anonymity of the system and increases the amount of trust users need to place in the operators. One advantage however is that users are able to verify the outcomes of their bets, thus preventing the operators from acting dishonestly. By using the client *seed* and parts of the blockchain transaction in generating the outcome, the operators cannot just generate failing bets in an attempt to deceive their users. This increases the trust between the users and the betting operators because it provides users with concrete evidence to use in a dispute over the outcome of a game.

Early on in its release, *Satoshi Dice* were open to *double-spend* attacks. They would process bets without waiting for any block confirmations in the *Bitcoin Blockchain*. This was because the bet outcome transaction would use the output of the original bet transaction, meaning that any block chain that did not contain the original bet, would also not contain the resulting payment. This design was flawed however, because once a bet was processed and the result instantly made known, if the bet was a losing bet users would quickly create a double spend transaction with a high transaction fee. They would then broadcast this to the network in the hope that the double spend would be confirmed before the original bet. Users would effectively use this to game the system, and to increase their chances of winning. They would let winning bets confirm but double spend losing bets. Because of this, *Satoshi Dice* changed their policy to wait for atleast 1 block confirmation before processing any bet. This made the attack more difficult to perform as attackers would need to create 2 blocks in the time the entire network has to create 1, making the attack much less likely. This is a great example that highlights just how important it is to wait for block confirmations in the network before processing bets.

2.3.5 Proof Of Existence & Gradbase

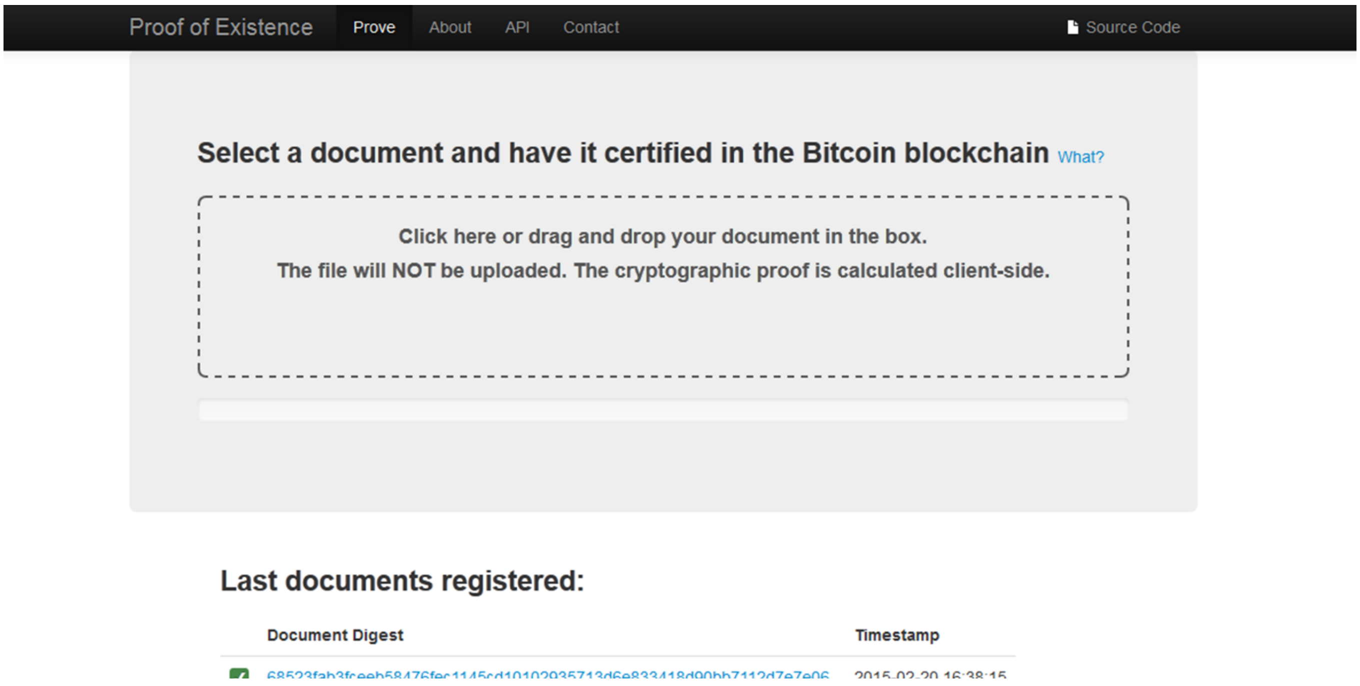
Proof of Existence [85] and *Gradbase* [7] are two interesting examples of *Bitcoin* applications that embed data directly into the *Bitcoin blockchain*. By using the blockchain as a storage mechanism they make use of several of the attractive properties that the distributed ledger provides, such as high replication, fault-tolerance and the *proof of work* mechanism. This allows them to store, timestamp and verify the existence of data, entirely anonymously and from anywhere in the world.

Proof of Existence uses the blockchain in order to prove the existence of a document or file. Using their service, you can anonymously and securely embed the cryptographic digest of any piece of data into the blockchain. This allows you to prove that a specific piece of information existed at a certain time. This can be useful for copyrighted material, proof of ownership and document integrity. *Proof of Existence* do this by providing a web application that allows users to hash various files on their computer and upload the hash of those files to the application. A unique bitcoin address is then generated for that user, allowing them to make a payment to that address in return for their hash to be embedded in the blockchain. *Proof of Existence* use the *OP_RETURN* script tag to embed that data into the blockchain and then provide an easy way for users to check whether or not a specific hash exists in the chain.

One advantage of this application is that you don't need to reveal anything about the actual data being hashed. Instead you provide the hash directly to the application. This means that the data is not left exposed or made publicly visible. Furthermore, hash functions are designed to contain a property called the *avalanche effect*, where a very small change in the input results in a large change in the output. This means that the very slightest change in the document or file being certified would result in a completely different hash. This allows the owner to ensure that the data being certified is not later open to change, i.e. somebody couldn't change the document slightly and have it result in the same hash, arguing that the original document was different.

Whenever a transaction is embedded into the blockchain, that transaction block is timestamped and is used in the *proof of work* for the next block. This means that every block confirmation strengthens the existing timestamps, effectively providing proof of data existence at a particular time. This service therefore can not only prove the existence of a file, but also provide an earliest estimated time as to when that file did actually exist. In addition to this, because the file hash is embedded directly into the decentralized ledger of the network, it also means that if *Proof of Existence* were to be closed down or later become corrupt, this would not affect the existing file certificates. Meaning users would still be able to certify the documents already embedded in the chain and do not need to rely on a single point of failure.

One of the disadvantages of *Proof of Existence* is that if the owner losses the original file, they have no way to recover it. The embedded hash cannot be used for data recovery and therefore becomes meaningless as the proof no longer works. Furthermore, this scheme is also vulnerable to a *collision attack*. If an attacker can construct some data that hashes to the same hash embedded in the blockchain they could use this as evidence against the original proof. Whether or not this would dismiss the original proof however is debatable as the likelihood of the constructed data making sense (i.e. being as readable as an English document) is extremely low. This is nonetheless a concern and is more a limitation of hash algorithms than anything else. Figure 16 shows the *Proof of Existence* website.



Select a document and have it certified in the Bitcoin blockchain What?

Click here or drag and drop your document in the box.
The file will NOT be uploaded. The cryptographic proof is calculated client-side.

Last documents registered:

	Document Digest	Timestamp
✓	68523fab3fceeb58476fec1145cd10102935713d6e833418d90bb7112d7e7e06	2015-02-20 16:38:15

Figure 16: Proof of Existence Website [85]

Similarly to *Proof of Existence*, *GradBase* uses the blockchain to verify degree certification. They provide a service where universities are able to publish certificates to graduating students. These certificates are hashed and embedded into the blockchain, and employers are able to perform background checks on students, verifying the credentials students claim to have. This works in a similar manner to that of *Proof of Existence*. When a university issues a certificate, they create and publish transactions with the destination address set to the hash of that degree certification. Employers can then use *GradBase* to check degree certifications by supplying the students credentials and looking for any blockchain transactions that validate those certifications.

This works by having a trusted mapping of public keys to universities, where the universities publish these transactions using their trusted public key. *GradBase* are then able to see the publications made by the various universities and use these to check students credentials. This model has many of the advantages that *Proof of Existence*, above, has. It is not centralized, meaning if *GradBase* is ever corrupt or shut-down, this does not invalidate the already published certificates. Furthermore, by storing the data in the blockchain you provide a very open and auditable way to verify honest operation of the parties involved, as you can see the information needed to perform these verifications directly in the blockchain.

One of the disadvantages of this approach is that the mapping of public keys to universities needs to be maintained. This means that either these keys need to be publicly published online (e.g. under the university website) or they need to be pre-agreed by both parties involved. What's more, the safety of the corresponding private keys need to be ensured, as exposing a private key, or having it stolen, could result in false certificates being published online or the invalidation of previously genuine ones. This may result in students being accused of falsifying their information, or people claiming to have qualifications they do not actually have. Figure 17 shows the *GradBase* logo.

Figure 17: GradBase Logo [7]

2.3.6 Reality Keys, Orisi & OpiDoki

Reality Keys [70], *Orisi* [92] and *Opidoki* [91] are three examples of applications that use *Bitcoin oracles* to determine the outcome of various events and statements. A *Bitcoin oracle* is a *Bitcoin* account or authority deemed to have jurisdiction over some piece of information, such as over the outcome of an event. *Reality Keys*, for example, provide a *Bitcoin oracle* service for various facts. Using their service, you can register to track an event, such as the outcome of a sporting event, and they will issue 2 public keys, one corresponding to the result of *yes* and another to *no*. When the event is finished, the private key corresponding to the resulting outcome will be published online and can then be used to sign various transactions or settle various bets. The private key of the outcome that did not occur is never released.

Reality Keys follow a database of facts and provide API access to their services, allowing you to build various markets and applications around truth statements. In the event that *Reality Keys* incorrectly publishes a result, users can ask an employee to manually double-check that outcome, usually for a fee. This allows various *Bitcoin contracts* and infrastructures to be built using their service. For example, *Reality Keys* might be used to issue promises between friends, where one friend promises to do something and offers to stake some amount on that promise. If they keep their word, their stake is returned to them, but if they fail to do what they promised, their friend gets to take the stake. This could be done by creating a multi-signature transaction requiring *2 out of 3* private keys to be spent. This transaction would take as input an amount that represents the stake. The three keys requires to spend it could be the private keys of the two friends and the outcome key released by *Reality Keys*. In the case that the promising friend kept their word, they could sign the transaction using their own key and the corresponding private key released by *Reality Keys*. This would allow them to regain access their funds. If the promise was not kept however, the other friend could sign the transaction using their own key and the opposite outcome key, therefore gaining access to the forfeited stake.

Orisi is another application that makes use of *Bitcoin oracles*. It is an open-source framework for constructing *Bitcoin contracts* around the outcome of various events. They do this through the use of several independent *Bitcoin oracles*. This has the advantage that instead of relying on a single oracle (like *Reality Keys*), you can choose to rely on a set of oracles, and construct an *m out of n* transaction, where atleast m of the trusted oracles must sign a transaction for that transaction to be spent. This has the advantage that bets are less susceptible to oracle corruption as an attacker would need to corrupt atleast m oracles to declare an outcome. This solution is therefore more robust to attack but it requires more oracles to be present in the system and to be willing to sign a transaction regarding a specific event.

Opidoki is a similar type of application to *Reality Keys* and *Orisi*, and is built on top of the *Counterparty* protocol. *Opidoki* provides an *oracle programming interface* where the outcomes of various events can be bet on. The way in which their solution works is that users are able to create an event by providing a snippet of javascript, a website url and a *final result time*. At the respective *result time* a website server will fetch the corresponding website url, run the given javascript on the webpage and broadcast the execution result in the *Counterparty* protocol. This will allow various bets and contracts based on the outcome to automatically settle in *Counterparty*.

One of the disadvantages of *Reality Keys* is that by using a single *oracle*, you make the system a single point of failure and target for attack. This means that if an attacker were able to corrupt *Reality Keys*, they would be able to incorrectly settle bets and publish untrue facts. *Orisi* overcome this problem by using multiple *oracles*, all who live in the *Bitcoin* ecosystem. Because *Orisi* require a transaction to be signed by multiple *oracles* before being spendable, communication needs to happen between them and this has to occur outside the *Bitcoin* network. As such, *Orisi* use *Bitmessage* [22], an alt blockchain to conduct this messaging and pass these transactions around. Unfortunately this adds additional overhead and complexity as you now need to store mappings between *Bitcoin* and *Bitmessage* accounts for each *oracle*.

Although *OpiDoki* do not have this problem, because they operate in the *Counterparty* ecosystem, their main disadvantage is that their solution is not very user friendly. The idea of placing a bet on the outcome of a javascript snippet and a url is not very attractive or trustworthy. It is not robust against human error, such as a mistake in the javascript code, and it is also vulnerable to network outages and attacks. Furthermore, because the entire system is built on the *Counterparty* protocol, it is not easy to see how these results are embedded into the blockchain. It therefore requires users to blindly trust the *Counterparty* and *OpiDoki* implementations, providing no ways to dispute the outcome of an event (e.g. if the javascript snippet is buggy, or the provided url is malformed.) In addition to this, *OpiDoki* requires users to have an account with *Counterparty* and some amount of *XCP* (*Counterparty's* internal currency). This adds further overhead and makes the entire process much more complex. Figure 18 shows the logos of *Reality Keys*, *Orisi* and *OpiDoki* respectively.

(a) RealityKeys Logo [70]

(b) Orisi Logo [92]

(c) OpiDoki Logo [91]

Figure 18: Logos of Various Bitcoin Oracle Applications

3. Legal Concerns

Due to the nature of this thesis it is worth paying attention to the potential legal issues and concerns that may arise when operating an anonymous *Bitcoin* betting exchange online. To ensure that we abide by any laws and legislation that may apply, we will briefly explore why these concerns are present. This is so that we can identify the potential areas of concern and lay out an appropriate course of action to address those areas.

3.1 Betting Legislation

As previously touched upon in the Background, section 2.2, online betting is a highly regulated and complex sector. Different countries around the world enforce strict and varying pieces of legislation regarding online betting and gambling. In the United Kingdom, where this thesis is being proposed, gambling is legal both online and off, provided the participants are over 18 years of age. We could think therefore, that our solution could be published online, as long as it included some form of age disclaimer or warning. This, however, is not the case, because our web application would make no distinction between users who are in locations where online betting is legal, and those where it is not. Publishing our solution online without enforcing some sort of location specific blocking would mean that anyone in the world would be able to bet, not just those in the United Kingdom.

Furthermore, because our solution operates anonymously, requiring no form of identification or registration from users, if any illegal activities were to occur, we would not be able to provide any useful information to law enforcement agencies. When you combine this with the fact that our application runs over an encrypted connection, using *https*, and that users may be employing some form of proxy or privacy tools to connect to the website, such as *TOR* [67], the only information that we could provide would be the public keys that were used to pay for the bets, information that is already available in the *Blockchain*.

3.2 Money Laundering

In addition to concerns about country specific betting laws, there are also other concerns that could arise if we were to naively host our solution online. As highlighted by Clare Chambers-Jones et al. in *Financial Crime and Gambling in a Virtual World* [129], page 155, "*the potential for digital currencies to be used to launder money is similar to that of virtual worlds and or gambling in general; they may provide a facility to place illegal funds and attempt to conceal their true origins. Similarly to virtual worlds, regulation is yet to be drafted, which may attract criminal activity*".

The possibility for criminals to use an anonymous *Bitcoin* betting exchange as a way to launder money is very real. When you consider that *Bitcoin* is easily exchanged for real fiat currency through various exchanges online, it is quite easy to see how this could occur. For instance, a criminal who wishes to buy something illegal online, using *Bitcoin*, might purchase some amount of *Bitcoin* through a fiat based exchange. Using this *Bitcoin* they may then try to use our betting exchange as a way to transfer the amount to an account that will be used to purchase the illegal goods. However, as the criminal does not want to be associated with that illegal account (e.g. by sending money directly to it) they might use the exchange as a way to cover their tracks. This could be done by placing a bet on the exchange, having it matched by the other account, losing that bet and having the total amount transferred to the illegal account. They could then argue that they had no association with the account and were simply placing a bet online. Although this might seem difficult to do at first, given that users are not aware of who their bets have been matched against before a bet is settled, a criminal might construct a very unappealing bet, at unattractive odds, that nobody would ever to take, except for the account that they wished to transfer from. This could then be used to guarantee that the two accounts are matched and that the transfer is successful, assuming the bet is settled the way that they intended.

Money laundering and criminal activity are very common in the *Bitcoin* world, with many high profile and serious cases having emerged within the last few years. One example is that of *Silk Road* [58], an anonymous black market that allowed users to illegally purchase drugs online. *Silk Road* ran as a *Tor hidden service*, allowing users to anonymously and securely interact with a website without threat of traffic monitoring. In 2013 *Silk Road* was shut down by the *FBI* and its owner, Ross Ulbricht, was arrested. During the investigation large amounts of *Bitcoin*, worth over $34.5 Million US, were seized by the *FBI.*

Unfortunately, high profile cases such as *Silk Road* are not isolated, and many other *Bitcoin* services, such as *Mt. Gox* [82] and *Liberty Reserve* [107], have been shut-down due to involvement with illegal activity and money laundering. In fact the case involving the *Liberty Reserve* was described by the Department of Justice as the *"largest international money-laundering prosecution in history"* [81], worth around $6 Billion US. This shows us just how attractive and popular *Bitcoin* has become for criminal activity, due to its anonymous and unregulated nature.

3.3 Our Response

In response to these legal concerns, we have decided to run the *Bitcoin Betting Exchange* online using the *Bitcoin* test network. The *Bitcoin* test network, also known as the *Testnet* [126], is a blockchain run by the developers of *Bitcoin* that allows users to test various *Bitcoin* applications. The difference between the *Bitcoin Testnet* and the real *Bitcoin* network, also known as the *Mainnet*, is that *Testnet* coins have no value, meaning they cannot be sold or traded for fiat currency. By making use of the *Testnet* in this thesis and avoiding the real *Bitcoin* network, we can safely publish our solution online without fear of violating any laws that may apply. This is because any bets placed by users will be done so using *Testnet* coins and will therefore be worthless.

Aside from the difference in value between the two networks there are also several small implementation differences. These differences however do not affect the way applications interface with the network and thus make the test network ideal for testing without fear that migration to the real network will cause unexpected behaviour. The difference between the two networks are that the *Testnet* operates on a different port, uses a different address header (to prevent *Testnet Bitcoin* addresses from being valid in the real network) and that it supports some non-standard transactions. *Testnet* coins can be freely accessed using various *faucets* found online [50].

One thing to note is that in order for the developers to ensure that *Testnet* coins have no value, they reserve the right to reset the blockchain at any time. This removes all transactions in the chain and clears the account balances of all *Testnet* accounts, setting them back to zero. This occurrence is rare, however, and has only happened twice in the history of *Bitcoin.* The first time was due to attempts from people to sell the coins for real money, and the second time was to fix several problems regarding block confirmation and waiting times.

4. Research and Design

In this chapter we outline and explore the potential solutions to the problems of *trust in an anonymous environment* and *private key security.* We identify various ways in which to approach these problems and discuss them in detail, making note of the benefits and limitations that apply to each one. We then introduce our proposed solution, explain how it works and analyse the advantages and disadvantages it provides.

4.1 Trust in An Anonymous Environment

Having analysed the related work in section 2.3, we observe that despite the large number of existing solutions in this space, we have yet to find a solution that addresses the issue of *trust in an anonymous environment.* All of the solutions we looked at require the user to pay for their bet by sending a payment directly to the *Bitcoin* account of the website or application. The website then acts as escrow, holding the user's funds until the bet has been settled and the winner paid. This means that the website has full control over the user's money once the payment has been received.

In addition, many of the solutions we looked at failed to provide any evidence that the user's payment had been received at all. Even the ones that did only provided a confirmation email, which by no means constitutes valid proof. This forces users to have to trust the service, and its operators, to act honestly and behave correctly.

4.1.1 The Bitcoin Network as Escrow

In the ideal scenario, a user would not have to trust a betting website or application with their money at all. Instead, the user and the application could enter into an agreement via a *Smart Contract* [120]. This would be a contract built using the *scripting language* primitives of *Bitcoin*, where the terms and conditions would be directly embedded into the user's payment. This would mean that the money sent by the user would only be spendable in the case that the appropriate conditions held. For example, just like you can only spend a *Pay-to-Public-Key-Hash* transaction using the correct private key for that transaction, the betting application would only be able to spend the user's payment if their bet lost.

A contract of this form would allow the bettor and the betting application to use the *Bitcoin* network as escrow, meaning that the funds would be provably unspendable until the bet was settled. In this case, the user would not have to worry about the betting application stealing their money, and likewise, the betting application would not have to worry about the user trying to take their money back. The network would act as ecrow, holding the funds until such a time that the contract could be resolved, at which point, either the user would get their money back and some winnings, or the betting exchange would get the money.

At a high level, an example of such a contract might appear as seen in figure 19: a user, Alice, wishes to place a bet that in the *2015 Wimbledon final Roger Federer* will win. A different user, Bob, wishes to match that bet as he believes that *Rafael Nadal* will win the final. The betting exchange would sit in the middle of the two users and when Bob matches Alice's bet, the exchange would construct a smart contract for the two, saying:

```
- If Roger Federer wins, Alice will get x Bitcoins.
- If Rafael Nadal wins, Bob will get x Bitcoins.
- Regardless of the outcome the Betting Exchange will get y percent of z Bitcoins.
- But, if something goes wrong, return the payments of Alice and Bob back to them
and terminate this contract.
```

Note that in this example z is the total amount of *Bitcoin* in the pot and x is the total amount minus the house edge of y percent.

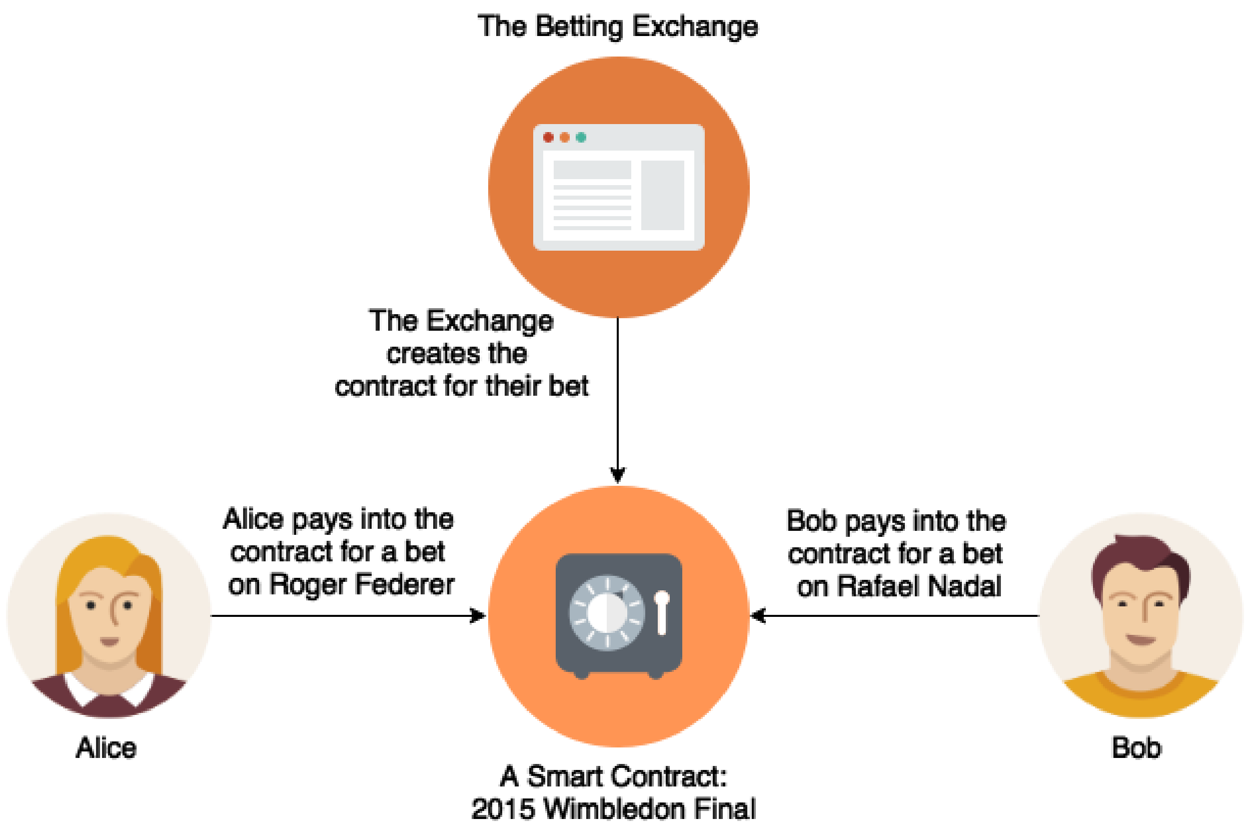

The Betting Exchange
The Exchange creates the contract for their bet
Alice pays into the contract for a bet on Roger Federer
Bob pays into the contract for a bet on Rafael Nadal
Alice
Bob
A Smart Contract: 2015 Wimbledon Final

Figure 19: Alice and Bob paying their bets into an address locked by a *Smart Contract*.

If such a construct were to exist in a *Bitcoin* script, this would be the ideal scenario for two reasons:

1. Firstly, the *Bitcoin* network would act as escrow, meaning that neither Alice, nor Bob, nor the betting exchange would have to worry about the other members stealing their money. This is because the transactions included in the contract would only be spendable under the conditions of that contract, meaning that no trust would be required between the parties.

2. Secondly, the betting exchange would not need to worry about storing these funds safely, as the transactions would be provably unspendable until such a time as the conditions were met. The only keys that would need to be managed in this scenario are the individual keys belonging to Alice, Bob and the betting exchange, and because the bet would be settled directly by the contract, the keys would not need to be stored online.

Although this would be ideal, there are multiple problems with this solution:

1. Firstly, how would the transactions generated by Alice, Bob and betting exchange be locked using a *Bitcoin script*? At present there are only 5 standard transactions supported by the *Bitcoin* network and it is not immediately obvious how those transactions could used to construct a contract of this form.

2. Secondly, how would the *Bitcoin* network be able to tell when, and if, any of the conditions in the contract held? For example, how would it know who won the *2015 Wimbledon final*, *Roger Federer* or *Rafael Nadal*?

4.1.2 The Data Outside the Blockchain Problem

The idea of using the *Bitcoin* network as escrow brings us to a fundamental issue currently facing *Smart contracts* and *Bitcoin scripts*. That is, accessing information outside of the *Blockchain*, through the *Bitcoin* network. *Gavin Andresen*, the chief scientist at the *Bitcoin Foundation* [52] and previous lead developer of *Bitcoin*, referred to this issue as the *"data outside the blockchain"* problem [4]. This problem is based on the fact that the *Bitcoin* network has a very limited amount of information available to it when executing *locking* and *unlocking* transaction scripts. This means that referencing external information inside a script is not currently possible [121].

In fact, this limitations is stated in the *Bitcoin Wiki* for *Smart Contracts*, *"Scripts are, by design, pure functions. They cannot poll external servers or import any state that may change as it would allow an attacker to outrun the block chain."* [121]. If Alice tried to spend the transaction that contained the contract, the *Bitcoin* network would have no way of knowing who won the *2015 Wimbledon final*. This means that building a *Smart Contract* using just the *scripting language* primitives of *Bitcoin* is not currently possible.

4.1.3 Third-Party Escrow

One way around this limitation is to use a third party escrow service. The idea here is much simpler and revolves around the concept of hiring an impartial third party to hold the funds of all bettors until the outcomes of their bets have been announced. In exactly the same way that an escrow works for the purchase of goods and financial deposits online, it would act as an impartial supervisor of the payments, only choosing to pay the winner when the result of the bet is known. Figure 20 illustrates the idea. Instead of Alice and Bob creating special *locking* scripts on their transactions, they would send their payments directly to a third party responsible for settling their bets. The third party would be told that these payments are for a bet on the outcome of the *2015 Wimbledon final* and that depending on who wins, the money should be paid to either Alice or Bob, with a small cut of the winnings to go to the exchange. When the event finishes, the third party would look up the winner of the event and proceed as required.

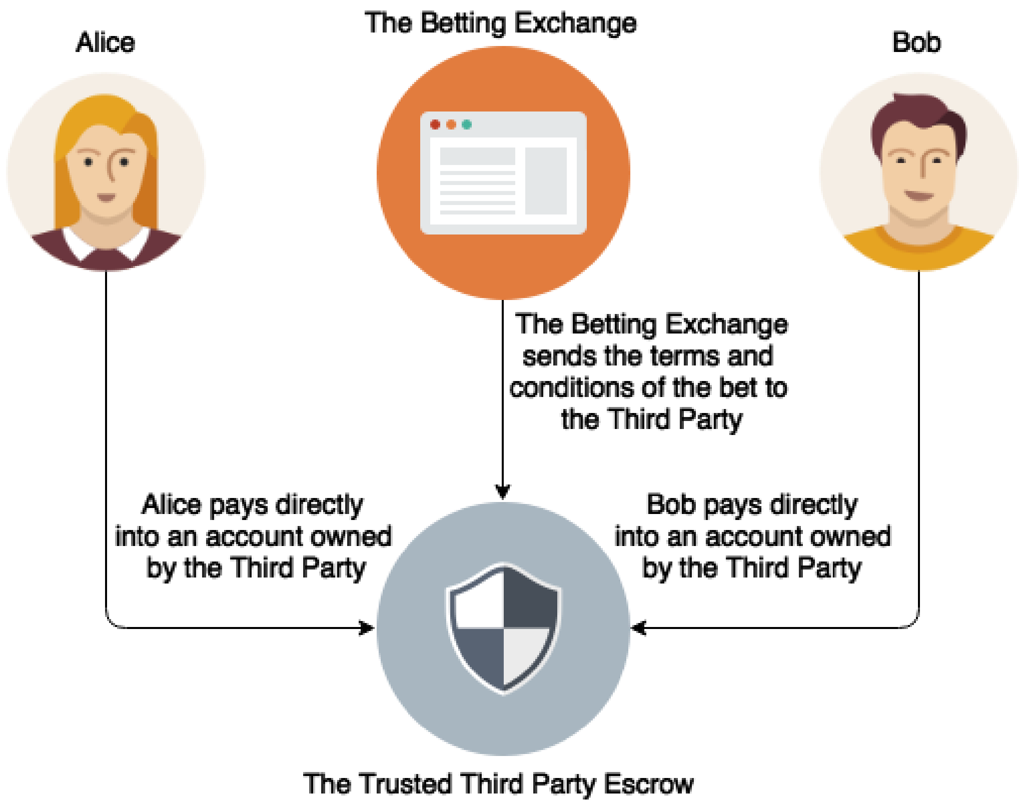

Alice
The Betting Exchange
Bob
The Betting Exchange sends the terms and conditions of the bet to the Third Party
Alice pays directly into an account owned by the Third Party
Bob pays directly into an account owned by the Third Party
The Trusted Third Party Escrow

Figure 20: Alice and Bob paying their bets directly to a third party escrow service. The betting exchange sends the terms and conditions for those bet payments.

Although this solution is definitely feasible, with several *Bitcoin* escrow services already operating online (such as *BTCrow* [30]), it too has several drawbacks:

1. The first is that this entire process happens manually. Alice, Bob and the betting exchange need to manually set up an *escrow transaction* in which the details of the contract are made known. Furthermore, the parties need to manually agree to the contract before it can be established. In addition, when the contract comes to be settled or executed, the third party needs to manually look up the result of the *Wimbledon final* in order to determine exactly how to settle the bet.

2. The second problem with this solution is having to pay a fee to the third party for their services. All of the existing escrow services available online require some form of fee to initiate an *escrow transaction*. Although this fee, around 1.55% [30], is minimal for many use cases, in a betting environment it adds significant overhead to the operation costs. One of the primary motivations for bettors to use a betting exchange over a sportsbook is that the odds provided are typically much more attractive (i.e. there is less overhead and bets can be matched directly between bettors). Employing a third party service to participate in every bet would significantly affect the odds the exchange is able to offer.

3. The final disadvantage of using a trusted third party is that nothing prevents the third party from stealing the funds either. Although you would try to reduce this risk by employing a well-known and trusted third party, the threat still exists. This is made worse by the fact that *Bitcoin* is much more attractive to steal because of its *anonymous* and *unregulated* nature. A third party in this situation does not provide much benefit because both Alice and Bob would have to trust it to operate honestly. It could be argued then that if they are both forced to trust someone, they may as well be forced to trust the betting exchange as this would limit the number of people involved, reduce the cost overhead and make the process simpler.

4.1.4 M out of N Transactions

One way in which to prevent a third party from stealing the funds is to use an *m out of n Bitcoin* transaction. Figure 21 shows how this would work. Rather then Alice and Bob making their payments directly to the third party or betting exchange, a *multisig* address would be constructed and they would send their payments directly to that address. This would allow a *2 out of 3 Bitcoin* transaction to be built, where the *multisig* address could only be spendable as an input when 2 of the 3 required keys have signed the transaction. The 3 keys in this example would be that of Alice, Bob and whatever third party they have decided to trust (e.g. the betting exchange).

By sending their bet payments to a *multisig* address it allows Alice and Bob to protect themselves from a corrupt third member while still allowing that third member to arbitrate between them. For example, if Alice wins the bet, she can create a *multisig* transaction that spends the funds provided by both herself and Bob (e.g. a transaction that spends the funds held by that *multisig* address). This transaction would include the percentage to be paid to the third member too, covering the costs of the betting exchange, for example. She can then sign her transaction and request that the third member sign it too. Now that atleast 2 out of the 3 participants have signed it, it becomes a valid transaction and she has been granted access to the funds. The same argument holds if Bob wins the bet. The third member cannot gain access to Alice or Bob's funds by itself, it requires the signatures of atleast one of them in order to spend a transaction.

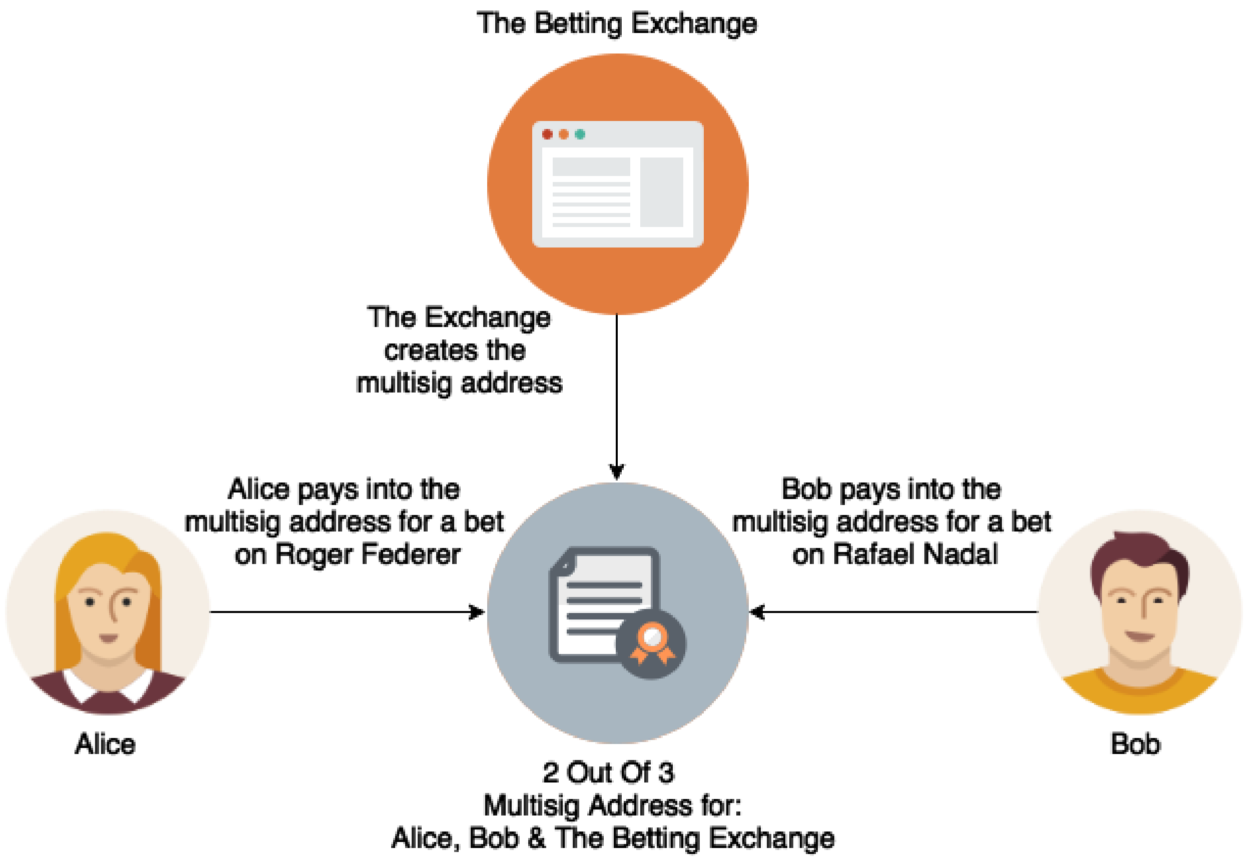

The Betting Exchange
The Exchange creates the multisig address
Alice pays into the multisig address for a bet on Roger Federer
Bob pays into the multisig address for a bet on Rafael Nadal
Alice
Bob
2 Out Of 3 Multisig Address for: Alice, Bob & The Betting Exchange

Figure 21: Alice and Bob paying their bets into a *2 out of 3 multisig* address. The betting exchange arbitrates between them.

This construct also allows Alice and Bob to work together if the third member becomes corrupt or simply refuses to co-operate. Note however, that it is still possible for two of the three participants to conspire with one another in order to spend the third participants funds. Although this is possible, we can again reduce this risk by selecting a well-known and trusted third member to arbitrate between them.

Proposing that this third member be a trusted third party outside of the betting exchange does not make much sense in this scenario. This is because here, the third party is simply acting as arbiter between the two bettors, to prevent them from stealing one another's funds. In this case, therefore, the best course of action would be to have the betting exchange act as the trusted third party, arbitrating between Alice and Bob. Not only does this reduce the number of people involved in the process but it also reduces the cost overhead, as now we do not need to hire an external third party.

Despite the benefits that a *2 out of 3 Bitcoin* transaction provides, there are still several significant disadvantages to this solution:

1. The first disadvantage is that in order for the participants to sign an *m out of n* transaction, this transaction needs to be passed between them outside the *Bitcoin* network. This means that some sort of infrastructure needs to exist that is external to the *Bitcoin* network in which messages can be passed between the various members. This is the same problem we identified when analysing *Orisi*, in the related work section, section 2.3. In order to overcome this, *Orisi* uses *Bitmessage*, an alt blockchain, to conduct message passing between the various parties. Unfortunately, requiring an external infrastructure introduces additional overhead and complexity, as mappings between *Bitcoin* accounts and *Bitmessage* accounts need to be maintained and communication paths made available. In addition, any sort of external system that need to be setup to allow message passing has the potential to negatively impact anonymity.
2. A second and perhaps more serious issue concerning the use of a *2 out of 3* transaction is the practicality of building a betting exchange around the transaction type. In order for Alice, Bob and the betting exchange to build a *multisig* address (an address to which Alice and Bob can send their payments), all three parties need to be known before the *multisig* address is generated. This is because in order to generate a *multisig* we require the n public keys of that *multisig* address (e.g. the n people who can sign it). This means that a bet match between Alice and Bob needs to have occurred before the address can be created and payments sent to it. In a betting exchange environment one member typically always places a bet before the other. The problem here therefore is that Alice and Bob can't pay for their bets until they know who they have been matched with, as the public keys of the bettors and the betting exchange need to be known in order to generate the *multisig* address.

These two disadvantages mean that interaction with the betting exchange from the user's side will be constrained and therefore need to look something like as follows: Alice comes to the betting exchange and registers her interest in placing a bet on *Roger Federer* for the *2015 Wimbledon final*. She doesn't pay for that bet yet, but she makes a note that she is interested in it and specifies her bet information. Bob then comes to the betting exchange and sees Alice's interest in the bet. Because Bob believes that *Rafael Nadal* will win the final, he decides to match Alice's bet. At this point both users are interested in the bet and a *multisig* address can be generated using Alice, Bob and the betting exchange's public keys. Unfortunately a problem now arises. We need to notify *Alice* and *Bob* that they can pay for their bets. Because you cannot notify *Bitcoin* addresses in the *Bitcoin* network, Alice and Bob will need to provide contact information outside of the network. Not only does this compromise anonymity, but it also significantly increases the amount of time taken for the parties to successfully create and pay for a bet. In this time, Alice may no longer be interested in the bet, the odds of the market may have changed, or the event already begun. Unfortunately, this user flow is not very practical given the constraints present in a *Bitcoin* betting environment.

4.1.5 Bitcoin Oracles

One attempt to solve the *"data outside of the blockchain"* issue is to make use of a *Bitcoin oracle*. A *Bitcoin oracle* is a *Bitcoin* account, or authority, deemed to have jurisdiction over some piece of information or state, such as over the outcome of an event. We have already seen several different *Bitcoin oracle* services when we looked at *Orisi*, *Reality Keys* and *Opidoki* in the related work section, 2.3. *Bitcoin oracles* try to make payment resolution much simpler by providing an API (application programming interface) to fetch external information that lives outside the *Blockchain*. This is done by making use of the *Bitcoin m out of n* transaction type, similar to what we have seen in the previous section.

Orisi for example, do this by providing a set of *oracles*, all of whom have knowledge about a specific event or outcome (e.g. *"Who Won the 2015 Wimbledon Final?"*). The idea is that when an event has finished, we ask each of the *oracles* for the outcome by giving them an *m out of n* transaction to sign. This transaction will be associated with a contract that was originally specified and so when asking the *oracles* to sign the transaction, they will only provide their signature if the terms of the contract hold.

For example, lets take a look at the case of Alice and Bob who wish to bet on the *2015 Wimbledon final.* Figure 22 shows how this might work. Just like in the previous example, once we have identified that Alice and Bob wish to bet against each other, we can construct a *multisig* address that references each of the *oracles* online. Once this address is generated, Alice and Bob will each send their bet payments to the address. When we generate this address we associate it with a contract (i.e. *"If Roger Federer wins, pay Alice. If Rafael Nadal wins, pay Bob... etc."*). This contract will be shared with the oracles when the *multsig* address is generated, so that the *oracles* will know whether or not they should sign any transaction that tries to spend the money held in the *multisig* address. So, in the case that *Roger Federer* wins the final, Alice will contact each of the *oracles* in turn and ask them to sign a transaction that she generated. Her transaction would spend the money stored in the *multisig* address, sending the majority to herself and some percent to the betting exchange. When the oracles receive this transaction, they will only sign it if the transaction conforms to the original contract that was associated with the address. Because these *oracles* have access to *sports data* they can see who won the event, and so they know that Alice should be paid the winnings, with a small portion going to the exchange. If all of the various conditions hold, each will sign the transaction in turn and Alice will be able to spend her winnings. If not, the *oracles* will refuse to sign the transaction.

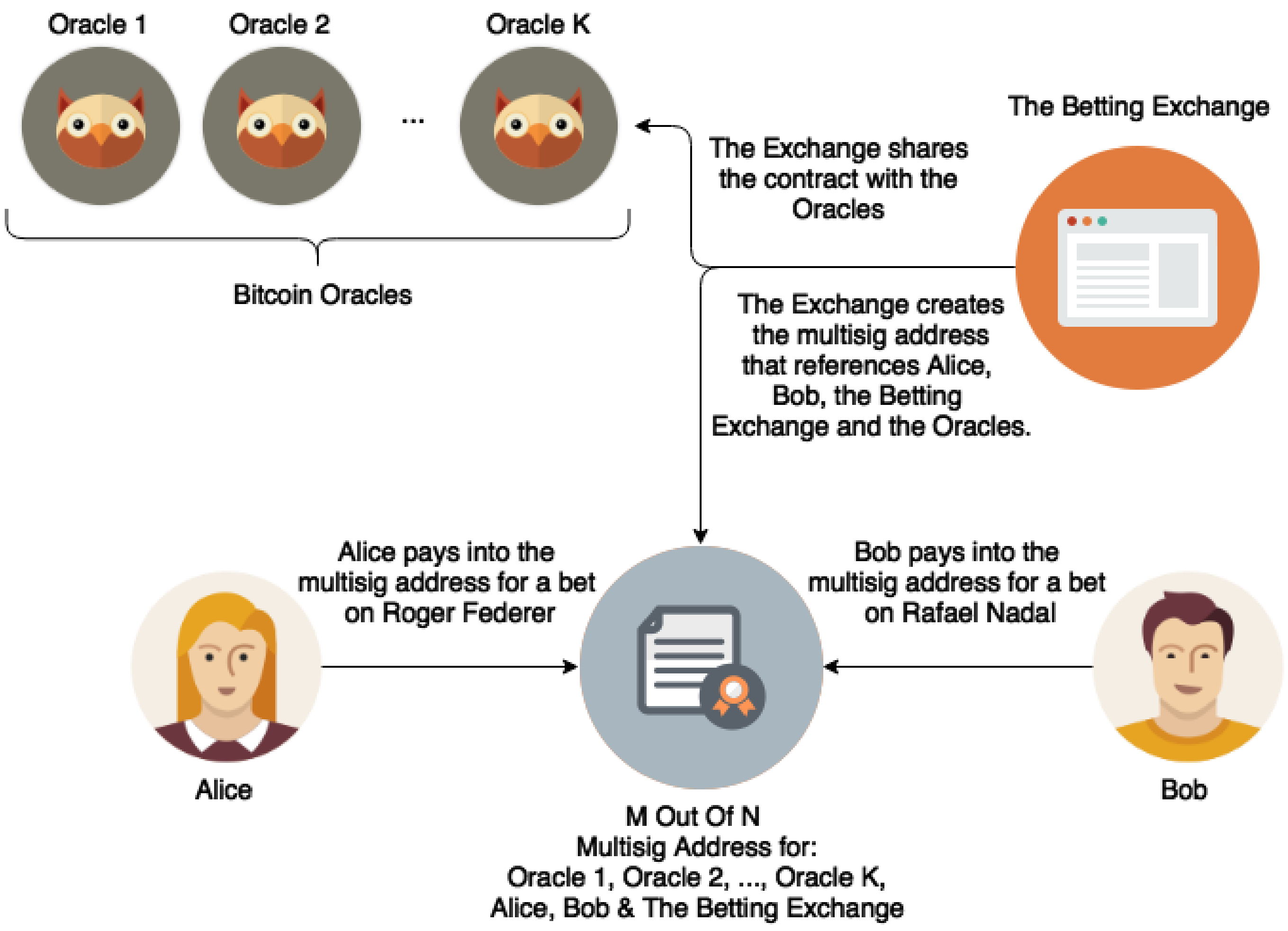

Oracle 1
Oracle 2
...
Oracle K
Bitcoin Oracles
The Betting Exchange
The Exchange shares the contract with the Oracles
The Exchange creates the multisig address that references Alice, Bob, the Betting Exchange and the Oracles.
Alice pays into the multisig address for a bet on Roger Federer
Bob pays into the multisig address for a bet on Rafael Nadal
Alice
Bob
M Out Of N Multisig Address for: Oracle 1, Oracle 2, ..., Oracle K, Alice, Bob & The Betting Exchange

Figure 22: Using a set of *Bitcoin oracles* to construct a *multisig* address and associating a contract with that *multisig* address.

Depending on how you set up the *multisig* address and what kind of parameters you assign the *m out of n* transaction, you can make the transaction specific to each bet. For example, increasing m and n will provide more protection from corrupt oracles, but make the signing process longer as more signatures are required. Likewise, if you want to factor in some sort of override, you can make m smaller and n larger, so that there are more possible ways in which to spend the transaction, trading off security for practicality.

Although *Bitcoin oracles* have many useful and interesting benefits, they all ultimately suffer from the same problem. This problem stems from that fact that they all rely on *m out of n* transactions, and as already mentioned, the construction of an *m out of n* address can only be done after the n public keys are known. Likewise, the contract that is to be shared with the *oracles* when the *multisig* address is created can only be specified after the bettors are known. We are therefore forced to identify the matching bettors before they can pay for their bets, and as discussed in the previous section, this restriction is not very practical in the case of a betting exchange.

4.1.6 Proof of Bet & Outcome

Due to the challenges and limitations currently surrounding *m out of n* transactions and *Bitcoin oracles*, we propose our own solution to address the problem of *trust in an anonymous environment*.

Instead of trying to remove the need for trust between the betting parties, as a *Smart Contract* does, we aim rather to minimize the amount of trust required between the parties. Our idea is to provide a *proof of bet* and *proof of outcome* system that allows third parties to verify the honest and correct operation of the betting exchange. By providing proof that a user has placed a bet, and proof that a bet has been settled, it means that if the exchange acts dishonestly (e.g. refusing to pay out winnings or incorrectly settling a bet), the bettor will be able to prove it to the rest of the world. This removes the need for users to blindly trust the exchange, as the proof would be enough to see exactly what happened and reveal which party was in the wrong. The same holds if a bettor makes false claims against the betting exchange.

It only takes one act of dishonesty from any of the participants to completely ruin their reputation. This is because the proof held by the other parties is enough to reveal exactly who misbehaved and is strong enough to prevent them from trying to argue against it. What's more, the proof also protects the reputation of the parties who behaved correctly and prevents false claims being made against them. In addition, the *proof of bet* and *proof of outcome* system that we propose also respects the anonymity of the bettors and their bets, and does not negatively impact or slow down the way in which users are able to interact with the exchange when placing a bet.

One might argue that a simple bet confirmation email or a screen-shot of the bet page would be enough to prove a user's bet. Emails, however, cannot be considered concrete evidence as they are easily forged. Furthermore, requiring an email address for each bet would compromise user anonymity. Likewise, screen-shots also cannot be considered concrete evidence, as they, too, are very easily manipulated. For example, someone might alter the *HTML* of their bet page and take a screen-shot of that, claiming to have won a bet. Or, alternatively, they might just alter the image directly using image manipulation software. In this situation, it would be the word of the user against the betting exchange and without concrete evidence it is very difficult to tell which party acted dishonesty.

Inspired by the way that *Proof of Existence* uses the *Blockchain* to timestamp the existence of documents, as discussed in section 2.3, we propose using the *Blockchain* to create, announce and store proofs. Whenever a bet is created by a user, we take the bet and the information given by that user, such as their wallet address, hash this information, and publish it into the *Blockchain* using the *OP_RETURN* transaction type. This transaction is embedded into the *Blockchain* using the *Bitcoin* account that belongs to the betting exchange. The idea here is that if a user has access to the original bet information, hashes it and proves that the resulting hash was embedded into the *Blockchain* by the betting exchange, then they can prove that any payment they sent to that address represents a payment for that bet. This process effectively allows a user to prove that the *Bitcoin* address they paid their bet into represents a bet payment. Similarly to *Proof of Existence*, this proof relies on the fact that a single change in the unhashed data produces a completely different outcome hash when compared to the original.

We can illustrate this concept by returning to the example of Alice who wishes to place a bet on the *2015 Wimbledon final.* Figure 23 illustrates this process in 3 steps. First, Alice goes to the betting exchange and selects to place a bet on the *2015 Wimbledon final.* In her bet she selects *Roger Federer* as the winner. At this point, in the second step of the process, the betting exchange generates a completely unique *Bitcoin* address just for Alice's bet and announces this into the *Blockchain.* The new address is owned by the betting exchange and will ultimately be the address that Alice sends her payment to. Now, before she sends her payment to this address, the betting exchange shows her that the address and the information of her bet have been publicly embedded into the *Blockchain* together. This means that if Alice sends any money to the new address, she can prove to the rest of the world that the money she sent represents a bet for *Roger Federer* in the *2015 Wimbledon final.*

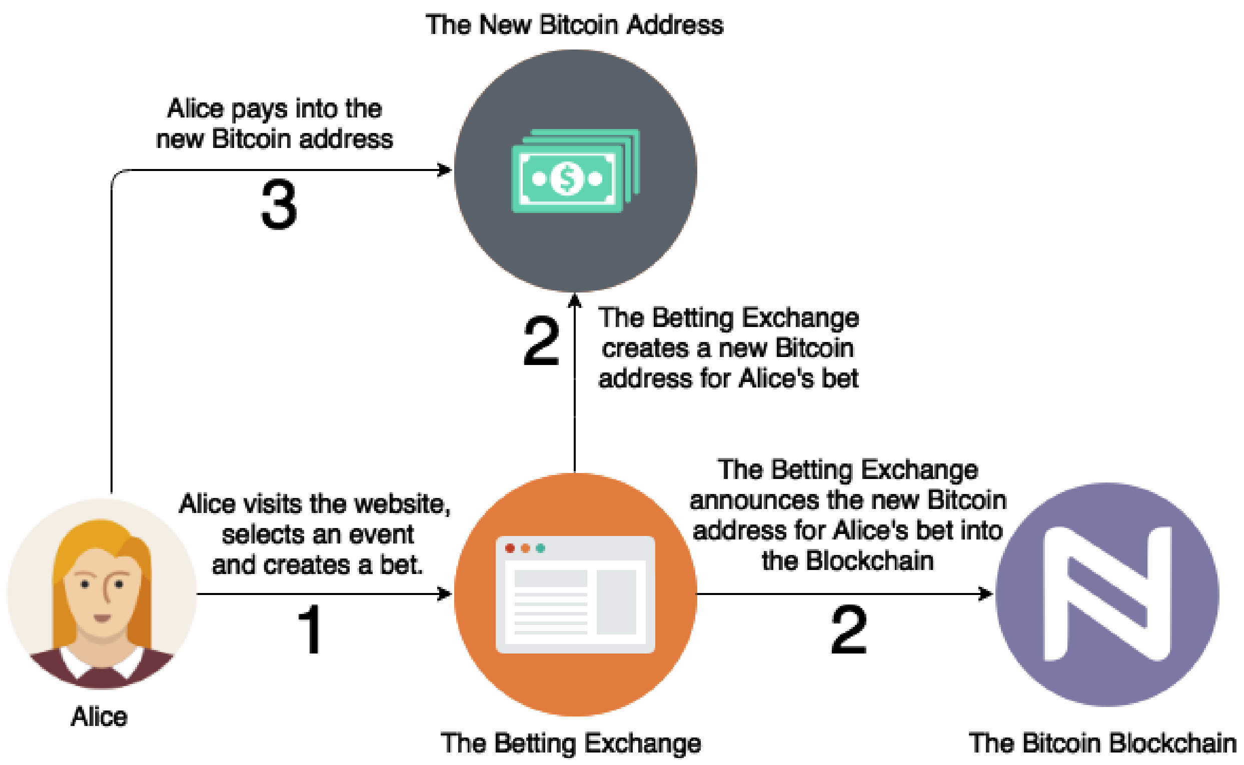

The New Bitcoin Address
Alice pays into the new Bitcoin address
3
2
The Betting Exchange creates a new Bitcoin address for Alice's bet
The Betting Exchange announces the new Bitcoin address for Alice's bet into the Blockchain
Alice visits the website, selects an event and creates a bet.
1
2
Alice
The Betting Exchange
The Bitcoin Blockchain

Figure 23: Illustration of how Alice would place a bet on *Roger Federer*.

Figure 24 illustrates how the process of announcing a new bet into the *Blockchain* works. When a bet is created, and before it has been paid for by the user, we hash the details of that bet (such as the event, the outcome and the user's address) and embed the resulting hash into the *Blockchain*. We do this by creating a new *OP_RETURN* transaction using the private key of the betting exchange. Seeing this hash embedded into the *Blockchain* and signed by the private key of the betting exchange, users are able to prove that any payment sent to the bet address is a payment for that exact bet (e.g. the event, market, outcome and odds in the data). Because these hashes are timestamped when they are embedded users can also show that the proofs existed in the *Blockchain* before any money was paid to those addresses. Note that we are implicitly assuming here that the public key of the betting exchange is considered common knowledge and that users know what public key to trust. This requires the public key of the betting exchange to be published somewhere safely online, for example under some form of trusted or signed manner (e.g. through a trusted *keyserver* [86]).

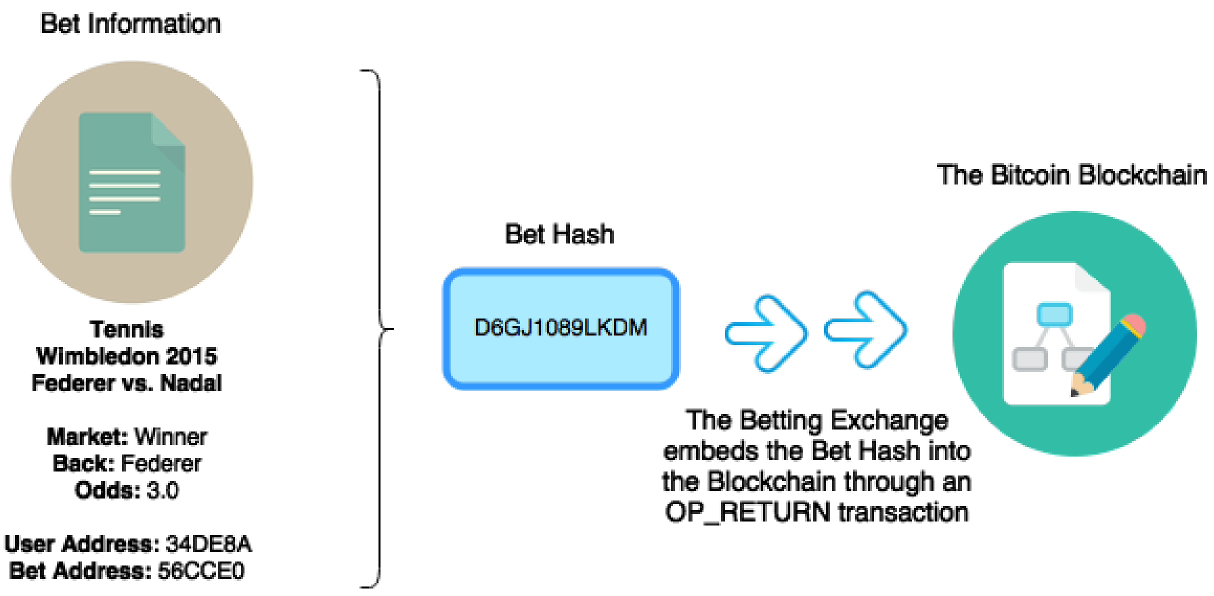


Figure 24: Hashing a bet and embedding it into the *Blockchain*.

Using *OP_RETURN* transactions we can automatically create new bet proofs whenever a user creates a new bet and allow them to verify it before sending their payment across. By embedding the proof into the *Blockchain* through a transaction, rather than sending it via an email, we have the advantage that proofs cannot be forged because a proof requires the private key of the exchange. Furthermore, proofs are also timestamped, globally accessible, stored on a distributed and replicated ledger and forever publicly available. Note that by embedding the hash of the bet and not the bet string itself, we still maintain anonymity for each bet. This is because a hash, by nature, is a one way function, meaning it is impossible for someone to recreate the original bet data from the hash. It is very easy, however, to verify that the hash represents the bet if you have access to the original data.

The *proof of bet*, when combined with the public transaction ledger, are enough for a user to prove the betting exchange dishonest. For example, imagine that a user places a winning bet and that the betting exchange refuses to pay them their winnings. They could prove this using just two transactions, the *OP_RETURN* transaction published for their bet and their payment. First, they present the *OP_RETURN* transaction created by the exchange and provide the data that hashes to the string embedded in that transaction. This proves that any payment to the bet address is a payment for that specific bet. Then they present their transaction that pays into the bet address, showing that the payment occurred after the betting exchange published the hash. They can then use the *Blockchain* to show that no transaction exists that spends their original payment and sends them their winnings (this would be in addition to some other transaction from the losing side of the bet, making up the total amount won). In this case the user has proven the betting exchange to have acted dishonestly.

There are three interesting observations we can make at this point:

1. The first is that this proof format works for any possible set of actions taken by the various participants. For example if the betting exchange pays the user less than they should have won, this too can be proven. The original payment transaction is public, and the data used to produce the hash includes the bet odds, so the user can use these two pieces of information to show that the actual payment is less than the expected payment. Likewise, if the user cancels their bet and their money is not returned to them, this too can be proven. Because all of the steps are auditable, dishonest behaviour by either party can be identified.

2. The second interesting observation to make is that if a dispute were to be raised and a user wanted to make a claim against the betting exchange, they would not have to release any personal information in order to do so. Instead, they would anonymously present the evidence and demand that the exchange takes an appropriate course of action (i.e. by paying them the correct amount). At no point does a user need to release any more information than is already available in the *Blockchain*. The only insight available to the outside world is that a specific *Bitcoin* address bet on a specific market, but the identity of whoever owns that address is not known.

3. The last interesting observation to make here is that when verifying these proofs there is only one step that needs to be performed manually. That is, looking up the result of the event and the market that the user bet on. Up until now we have assumed that results of events and markets are common knowledge (i.e. that everybody knows who won a match and what the final score was). This is because finding the result of a sporting event is generally considered simple when done outside the *Blockchain*. Results are available both online and off, and easily found through many different sources (e.g. news websites and television programs). Although looking up this information manually is relatively straight forward, complications can arise. For example, what if two sources seem to disagree on how to settle a market (i.e. because of some unusual circumstance). Or what if finding a reliable source is difficult because the event being bet on is not very well known (e.g. a local football game, or a small league match). In these cases, it slows down the verification process and weakens the proof, making it open to attack. For example, what if an attacker publishes a website that contains sports results, all of which are correct, except for the result they bet on. In this case, without looking at any other sources, it would appear that the betting exchange incorrectly settled their bet. Alternatively, what if the attacker instead changed the result of an event on a trusted website by injecting a piece of javascript into that page. Looking at this result without consulting other sources would also show the betting exchange to be dishonest. With so many markets and events being bet on daily, oversight is possible and mistakes can slip through.

In order to avoid issues with verifying the outcome of a bet, we can take our proposed idea one step further. Instead of requiring proof of outcome *outside* the *Blockchain* (e.g. through a link, or url) we propose that we make the outcome available *inside* the *Blockchain* (similarly to how we have already done for proof of bet). This will allow anybody who wants to verify the correct settlement of their bet to see the proof of outcome directly in the *Blockchain*, without needing access to third party websites or sources.

Our proposal is as follows: we assign the responsibility for declaring the outcome of a market to the official sports authority for that market. For example, in a tennis event, we assign the responsibility of announcing the results to the *Association of Tennis Professionals [87]*, the *ATP*. Likewise, for an international cricket event we assign the responsibility to the *International Cricket Council [40]*, the *ICC*. Doing this removes the ability for the exchange to announce an incorrect outcome and thus prevents them from being able to lie to users about who won in order to steal funds.

Similarly to how *Gradbase* assign a single public key to each university and trust it to announce valid certificates (see section 2.3), we can assign a single public key to each sports authority and trust it to announce valid market outcomes. When an event finishes, the sports authority can announce the winner of the event into the *Blockchain* by hashing the outcome they wish to announce and embedded into the *Blockchain* using an *OP_RETURN* transaction. Figure 25 illustrates how this works. The outcome is hashed, and the resulting hash is embedded into the *Blockchain*. This is very similar to how bet proofs are embedded into the *Blockchain*, except the public key announcing these *OP_RETURN* transactions belong to the sports authorities and not us. Because we only trust transactions that are signed by the public keys of the trusted sports authorities it means that even if somebody else were to embed these hashes in the *Blockchain*, they would not be able to settle these markets. This means that we do not need to worry about an attacker embedding these in an attempt to settle a market.

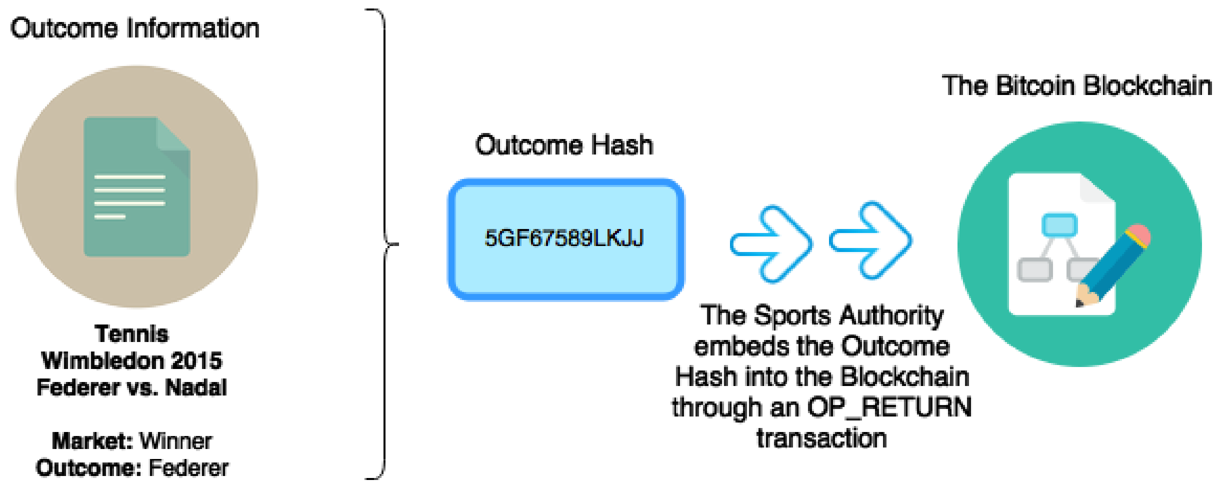


Figure 25: A sports authority hashing the outcome data and embedding it into the Blockchain.

One question to ask regarding this process is what exactly is the motivation for sports authorities to use our system and to settle these bets? Because sports betting and gambling make up such large proportions of revenue for many professional sports teams and events (e.g. through sponsorship and endorsement), providing a system that makes online betting more trustworthy and reliable for users will help promote these activities. This will not only make betting safer and more accessible to the average *Bitcoin* user, but also to the general population. In addition, because these sports authorities are already having to publish these results online manually anyway (e.g. through various websites), by providing a much more secure way for them to announce results they become less open to attack and inconsistency (i.e. they can have a single source of truth for event results that cannot be tampered with or altered). As such, we believe that our system would not only benefit *Bitcoin* users and bettors, but also the sports authorities and teams surrounding the events.

One advantage of our solution is that it does not interfere with the way that users are already used to interacting with a betting exchange. They are not required to create special *locking* transactions, sign various contracts or contact external *Bitcoin oracles* in order to spend their winnings. They simply pay for their bet by sending their money to an address and their winnings are automatically returned to them. This is especially important for *Bitcoin* users who are not versed with exactly how the latest transactions work and instead just want to bet.

Furthermore, our solution is much more practical than that of a *Bitcoin oracle* or *m out of n* transaction. This is because bets can be paid for immediately when they are placed and matches can happen instantly. No users have to be contacted or notified when they can pay for their bet and the wait time between potential matches is much lower. In addition, our solution does not compromise user anonymity and does not require any personal information from the user other than the address of where to send their winnings.

Another advantage of our solution is that by assigning the publication of event outcomes to the sports authorities responsible for that event, we prevent the betting exchange from being able to declare incorrect or false event outcomes into the *Blockchain*. This prevents them from being able to act dishonestly in order to incorrectly settle a bet.

One disadvantage of our proposed solution however, is that it is still possible for the betting exchange to steal money. For example, it is still able to refuse to pay a user. As soon as it does so however, the user will be able to prove it and the reputation of the betting exchange will be destroyed. This means that no users will ever trust the site again. Furthermore, this proof cannot be disputed against by the betting exchange, as it clearly reveals which party was in the wrong.

Another disadvantage of our solution is that event outcomes have to be manually embedded into the *Blockchain* by a sports authority. Although manually announcing results is not ideal, you could argue that sports authorities still have to do this anyway. For example, *Bitcoin oracles* require some form of data feed or API in order to know the outcomes of various events. Because these data feeds are ultimately being populated manually at the source (e.g. by the sports authorities through their websites), the amount of work required does not change. Furthermore, our solution actually provides a much more secure way for them to announce results when compared to that of a website. For example, websites can easily be attacked, altered, brought down or corrupted. Announcing results through the *Blockchain* instead provides a single source of truth that is distributed, globally accessible, timestamped and cannot be altered or changed.

4.1.7 Bet Verification Check

By combining the *proof of bet* and *proof of outcome* mechanisms we saw above, with the information in the public transaction ledger, we can construct a *bet verification check* that ensures the correct settlement of any bet. Because the *proof of bet* contains the exact details of a users bet and the public transaction ledger shows all incoming and outgoing payments for that address, we can tell exactly how much was placed on a specific outcome at what odds. Furthermore, when you combine the *proof of outcome* declared by the trusted sports authorities to this information, we know exactly how much the user should be paid.

By monitoring all incoming and outgoing payments to the unique *Bitcoin* address that represents the user's bet, we can see exactly how much was initially paid, how much was returned (for the unmatched portion), and how much was won. By combining all of this information with the proofs, a user can not only verify the correct settlement and appropriate payments for their bet, but any third party with access to the information can do so as well. We call the process of verifying this information for a single bet a *verification check*. This check has several interesting and useful properties that we can later exploit when addressing the concerns of *private key security*. In addition, this *verification check* also allows a third entity who wishes to audit the exchange to do so without requiring the exchange to explicitly search for and collect all the appropriate information.

4.2 Private Key Security

Our solution to the problem of *trust in an anonymous environment* requires the betting exchange to securely manage and store many public and private keys. These keys give access to the various *Bitcoin* accounts that users send their bet payments to and so careful consideration needs to be given to the way in which they are managed. If an attacker can get access to the private keys stored by the exchange, they will be able to steal the money held in those accounts. This will not only be a financial catastrophe but it will also severely damage the reputation of the betting exchange. Because of the sensitive nature of private key security, the majority of the betting websites and applications that we analysed do not openly discuss how their keys are stored and secured. As such, it is difficult to analyse the specifics of many of the existing solutions. We can however, pay close attention to the way in which we architect our own solution in order to minimize security vulnerabilities and push back against various threats.

4.2.1 Generation of Keys

The first issue we need to address when thinking about private key security is key generation. It does not matter how securely we store our private keys if they can ultimately be generated by someone else. This is because *Bitcoin* private keys are essentially just 256-bit numbers in the range of:

`0x1`, to

`0xFFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFE BAAE DCE6 AF48 A03B BFD2 5E8C D036 4140`

Or in decimal format:

`1`, to

`115792089237316195423570985008687907852837564279074904382605163141518161494337.`

Almost any 256-bit number is a valid *ECDSA* private key (*Elliptic Curve Digital Signature Algorithm* private key). This means that every possible private key is known because it is just a number within a very large range of possible numbers. In fact, *directory.io* [47] lists every possible private key imaginable!

What makes private keys secure is the randomness in which they are selected. Because the key space is so large, in the order of 2^{256}, the chance of two users selecting the exact same private key is exceedingly small, so much so that it is almost inconceivable. In addition, because private keys are kept secret and not derivable from public keys, it is not possible to tell which private keys have already been selected and which haven't. It is therefore almost not worth worrying about another user selecting the same private key as you, assuming a truly random and evenly distributed selection of keys.

Unfortunately, however, random number generators that are typically used to generate private keys are not truly random. Instead, they are *pseudorandom*, deterministic upon some seed. Although many of these algorithms, when appropriately seeded are good enough for purpose, weaknesses can arise from insufficiently seeded generators, incorrect implementations and mistakes. It is therefore important to pay attention to the way in which these private keys are selected. For example, by appropriately seeding random number generators and using trusted and well-known implementations we can reduce the risk of this happening.

One example that highlights the weakness in deterministic key selection is a *Bitcoin Brainwallet* [118]. A *Bitcoin Brainwallet* operates using the idea of storing *Bitcoin* private keys by memorizing a passphrase. It works similarly to that of storing a password in one's head, rather than writing it down on a piece of paper, or in a file. *Brainwallets* are able to take a passphrase, and deterministically generate a private key from that passphrase. The idea is that, if the passphrase is not written down, the private key and funds stored in the account remain secure. When a user wants to make a payment or use their account, they will deterministically generate their private key from their passphrase, use it to sign a transaction and then safely discard the private key. This means that private keys are not physically stored anywhere but generated using the passphrase that has been memorised.

Despite the obvious disadvantage that if you forget your passphrase you permanently lose access to your funds, *Brainwallets* are actually very insecure. This is because the passphrases that humans typically choose to remember relate to various aspects of their life, such as quotes from books, movies, or names of places. This means that the key space by which to choose private keys actually becomes significantly smaller, and so people typically end up choosing passwords and phrases that can be guessed through trial and error (e.g. brute force). For example, by using simple a *dictionary-attack* [101] you can search through many well known words, phrases and quotes in order to find private keys that have already been used. In fact, many dictionary attacks have already been performed on several brain wallet services online, with concerning results [94].

The point to take away here is that in any *Bitcoin* related service or application, attention needs to be paid to the way that private keys are generated.

4.2.2 Backing up Keys

In addition to private key storage, one element of *Bitcoin* security that is often overlooked is that of maintaining private key backups. One of the limitations of using *Bitcoin* when compared to a centrally government currency is that if you lose your private keys, you have no way in which to regain access to your accounts. Unfortunately, this means that any money held in those accounts are forever lost. For example, there was one case that recently occurred where someone threw out an old computer hard drive that had been used to store their private keys. Losing this hard drive meant that around 7500 *BTC* were permanently lost. This had an estimated value of £4.6 million [9] at the time.

These types of losses are actually quite common, and estimates have been published online stating that around 30% of all mined *Bitcoins* are in fact, *zombie bitcoins*, *Bitcoins* belonging to private keys that have been lost. This equates to a total of £625 Million [109] in losses. As such, it is important to maintain regular backups of private keys in order to prevent such catastrophes.

There have been several interesting developments in the last few years regarding regular backups of *Bitcoin* wallets. One approach is to make use of *hierarchical deterministic wallets.* As previously touched upon, *hierarchical deterministic wallets* generate new private keys based upon a seeded value. The idea is that generation of these keys is deterministic upon that seed, and means that in order to backup all of the private keys one owns, they only need to backup the seed. *Bitcoin Trezor*, for example, makes use of this. Although this is beneficial for backup purposes it means that anybody who knows the seed can generate all of the private keys for that seed and so in one sense it might suffer the same vulnerabilities as that of *Brainwallets* above, if the seed is not appropriately chosen.

Other approaches to secure private key backups include copying the keys and storing them in some form of encrypted or secure manner, printing them out and storing them in a safe location, or simply creating a snapshot of your wallet application every few weeks.

Of course, regardless of how you choose to backup your private keys, specific attention still needs to be given to the security of those backups. By nature, a backup prevents you from losing your private keys, but that means that whoever can get access to your backup has access to your keys. As such, the same amount of attention that is given to the security of key storage needs to be given to the security of backup storage.

4.2.3 Storage of Keys

Storing private keys securely is perhaps the most important aspect of any *Bitcoin* based web application or service. This is because typically, private keys need to be accessible by the application in order to send and receive payments and interact with the *Bitcoin* network. Depending on the requirements surrounding the application there are many different ways to store keys, and each method has its own advantages and disadvantages.

One primary characteristic that can be used to split up many of the existing storage techniques is whether or not the keys need to be accessible online or offline. Online storage is typically referred to as *hot* storage because the keys are available to some application or process that is connected to the Internet. Offline storage however, is often referred to as *cold* storage because the keys do not need to be available a service running online. The advantage of *cold* storage is that users can physically secure their keys by storing them in a vault or safe, or by hiding them in an appropriate place. This means that if an attacker wanted to gain access to their funds, they would physically need to locate and steal those keys.

Cold storage by nature tends to be more secure, because private keys do not need to be accessible at all times. Examples of *cold* storage include *paper wallets*, where private keys are written on a piece of paper, and hardware wallets, where the keys are stored on a hardware device that is not connected to the Internet. Figure 26 shows an image of a *Bitcoin* paper wallet that can be printed on to a piece of paper and stored in a safe location. Encrypted storage of keys online can also be considered *cold* storage if the encryption key required to decrypt the *Bitcoin* keys is stored offline.

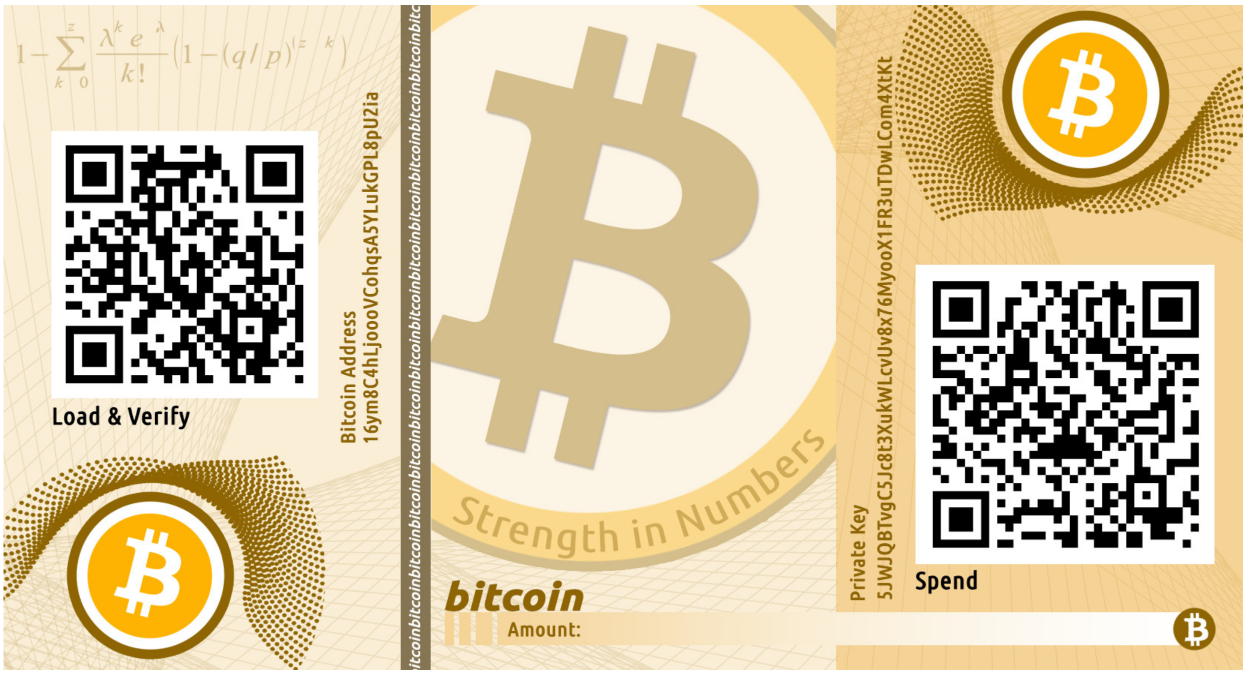

Load & Verify
Bitcoin Address
16ym8C4hLjoooVCohqsA5YLukGPL8pU2ia
Strength in Numbers
bitcoin
Amount:
Private Key
5JWJQBTvgC5Jc8t3XukWLcvUv8x76MyooX1FR3uTDwLCom4XtKt
Spend

Figure 26: A *Bitcoin* paper wallet.

In contrast, *hot* storage is required where the keys need to be accessible to a running service or application, as is the case with the majority of betting websites. *Hot* storage, or wallets, are often more vulnerable to attack than *cold* storage. This is because the keys and accounts are being used and accessed in a continuous manner. For example, whenever a user places a winning bet on a betting website, the application needs to pay that user in real time and so must be able to access the private keys when needed. As such, a vulnerability that allows an attacker to gain access to a web server, or read its local storage, will allow that attacker to gain access to the keys.

Examples of *hot* storage include physical storage on a webserver, where the private keys are simply written to disk, or a database, where the keys are held in a structured format. There are of course various ways in which to increase the security of private keys held in *hot* storage. Good security practices, for example, encourage the use of additional security layers to make it more difficult for an attacker to read the keys. Encryption, for instance, is one example of an added security layer. By not storing the keys in plain-text, it means that an attacker would need the decryption key in order to read the private keys. Of course, the application would also need access to the decryption key as well, so the attacker could just monitor the application to see how it gets access to that key. In general, however, security layers make it more difficult for malicious parties to perform these types of attacks.

In addition to *hot* and *cold* storage, you can also use a *multi-tiered* approach when storing private keys. A *multi-tiered* approach makes use of both *hot* and *cold* storage in order to regulate and limit the amount of money that is most vulnerable to attack at any one time. Figure 27 shows a diagram of a *multi-tiered* architecture. As can be seen in the diagram, we have a betting application that communicates directly with a *hot wallet*. It uses this hot wallet to receive payments from users and send payments to users, and so it has access to all of the funds in that wallet. However, when the amount of money in that wallet increases above a specific threshold, the application sends the excess money to a *Bitcoin* account held in *cold* storage. This reduces the amount of money that is vulnerable to a *hot* wallet attack at any one time. Likewise, whenever the amount of money in the *hot* wallet becomes less than a certain threshold, the operator of the website will manually send a payment from the account in *cold* storage to the account held by the *hot* wallet. This idea is similar to how a bank or shop will limit the amount of money held on the shop floor at any one time. This is to guard against theft.

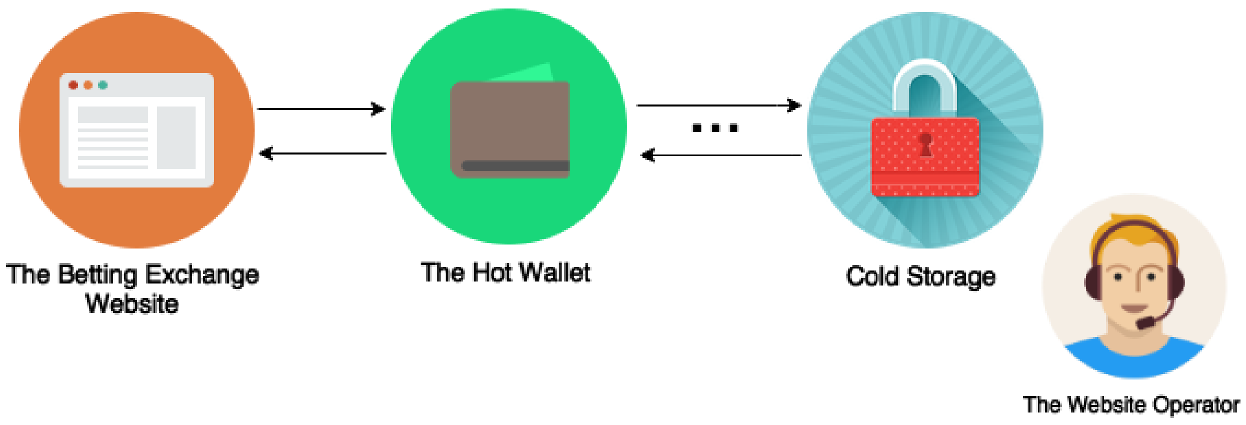


Figure 27: A multi-tiered approach to storing *Bitcoin* private keys.

The nice thing about this approach is that there can be more than just 2 tiers. Additional layers of storage can be added in between the *hot* wallet and the *cold* wallet in order to guard against other threats and provide further benefits. For example, you can add a slightly more secure hot wallet in between the two layers. This wallet would only be accessible from specific addresses, such as from internal traffic. Doing this provides a slightly more practical approach to topping up the hot wallet as it doesn't have to be done manually by the operator. In addition, the server would be slightly more secure than the first *hot* wallet because it is not open to general communication and can only be accessed on-site.

Another approach to storing keys is to use a hardware based private key store. Hardware based private key stores are specialized pieces of hardware that have been designed to operate in a very controller manner. This is in order to minimize and push back against the possible threats that arise from storing private key data. As previously mentioned, *Bitcoin Trezor* is one example of a hardware based private key store. *Bitcoin Trezor* physically stores the private keys on a hardware device, in a *cold* offline manner. It allows users to interact with it by connecting the device to a computer. *Bitcoin Trezor* operates in such a way as to never expose the private keys of the device to the computer and instead signs transactions internally, inside the device. This means that regardless of how insecure the computer is that it is connected to, it will not expose the private keys to the external environment. Furthermore, *Bitcoin Trezor* takes into account the fact that the user's computer may contain viruses, keyloggers and other malware, and so provides a secure way for users to interact with the device, without allowing the computer to make unauthorized requests.

One of the advantages of using a hardware based private key store is specialization. By physically building the device in hardware you can enforce certain characteristics in the way that the device operates (e.g. *Bitcoin Trezor* don't ever send the private keys to the computer). This limits the number of attacks that a malicious party can perform in order to gain access to the private keys stored on the device. As previously mentioned, *Bitcoin Trezor* uses a *hierarchical deterministic wallet* in order to generate new private keys. This means that should the device fail, you will still be able to regain access to your private keys if the seed has been backed up.

4.2.4 A Smart Private Key Store

Taking the concerns of private key *generation*, *storage* and *backup* into account, we propose our own solution to address the issue of *private key security* in a betting exchange. We do this by suggesting the design of a hardware based, smart, private key store, built specifically for our purpose.

Inspired by the design of *Bitcoin Trezor* we propose our own concept for a smart, private key store that makes use of *bet verification checks* to sign payments and settle bets. The smart key store performs these actions without ever exposing the private keys to the betting exchange or to the outside world. This means that under a threat model where an adversary has control over the web application, the private keys and funds for the exchange still remain intact.

We do this by creating a hardware device designed specifically for our needs. The hardware device, just like *Bitcoin Trezor*, will store the private keys of the betting exchange internally, physically on the device. The device will be plugged directly into the server that runs the web application for the exchange, and so will only be able to communicate with that server. In addition, the device will provide a fixed and limited set of communication functions in order to allow the betting exchange to operate effectively. Just like *Bitcoin Trezor*, the signing of transactions will only happen internally, inside the device, and so the private keys will never be exposed to the computer in which it is connected. Furthermore, in order for the computer to get the hardware device to sign a transaction, such as when a user has placed a winning bet and needs to paid, the web application is required to provide adequate proof that the signing request is genuine and valid. This is what makes the device *smart*.

Figure 28 illustrates this concept. As can be seen in the diagram, the smart, private key store is physically connected to the computer, or server, that runs the betting exchange. When Alice and Bob access the betting exchange via their browser, they do so by connecting directly to the server. The betting exchange web application communicates with the private key store through a physical connection (e.g. a usb connection) in order to generate new addresses and sign transactions.

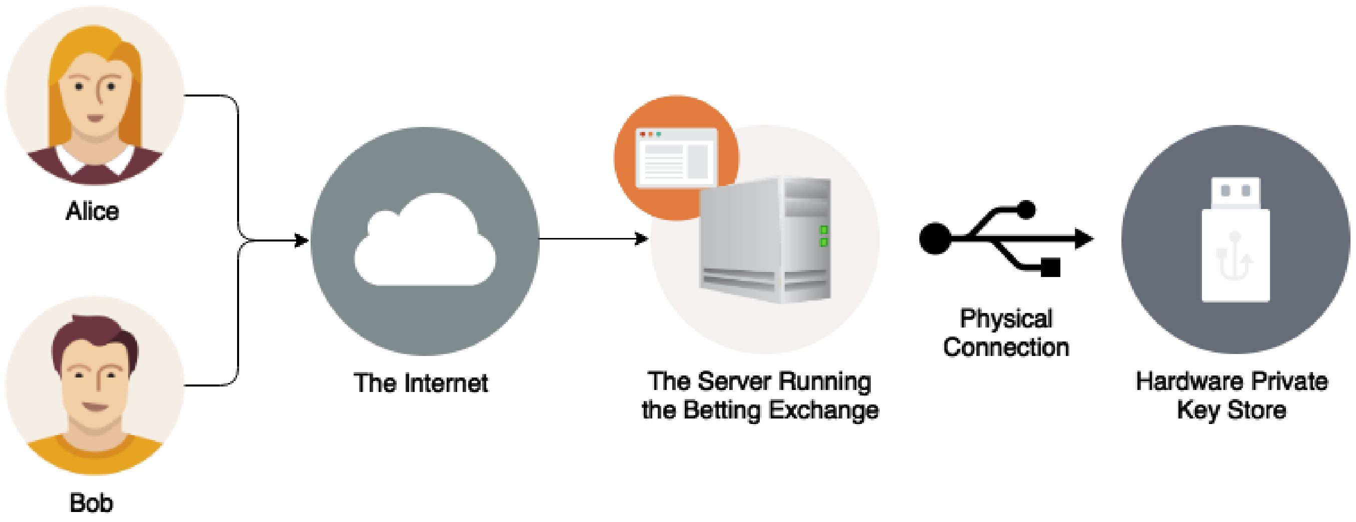

Alice
Bob
The Internet
The Server Running the Betting Exchange
Physical Connection
Hardware Private Key Store

Figure 28: A hardware based, smart, private key store connected to a web server that hosts the betting exchange.

We gain several advantages by implementing the key store as a hardware device:

1. Firstly, by designing the device to never expose the private keys to the outside world, it means that in order for an attacker to gain access to the keys, they need to physically read

the device's storage or memory when it is running. Given that the device will be connected to a web server and located in a room where the server is held, gaining physical access to it will be all the more difficult.

2. Secondly, by implementing the device as a hardware device, we have the advantage that under periods of significant attack, or when the betting exchange is not operational (e.g. under maintenance), the device can be unplugged and stored in a safe or vault, making it an effective means of *cold* storage.

3. Finally, by using hardware we are able to hard code, or *bootstrap*, certain pieces of information to the device to make it more secure. This could be through a one time setup, or by physically embedding the information into the device when constructed. For example, a shared secret or encryption/decryption key-pair could be hard coded into the device to make sure that it only ever communicates with a source that can provide that shared secret. This means that if the device were to be connected to another computer, it would refuse any attempts at communication unless that computer could authenticate itself by proving it has the shared secret.

In order to make key storage on the device slightly more secure, we can also encrypt the private keys before they are written to the device. This will prevent someone with physical access to the device from simply reading the private keys off it. Instead, they would now require the decryption key in order to decrypt the *Bitcoin* keys. The decryption key can be stored outside of the device and passed in to it as necessary, for example, when it needs to sign a transaction. Of course, an attacker could just wait for the decryption key to be passed into the device before extracting the private keys out of memory, but, if we minimize the amount of time that the decryption key is available in memory and encrypt the communication link, it makes this type of attack more difficult.

As described above, what makes the device *smart* is the way in which the server is able to interface with it. The device only supports a limited number of interface operations and so restricts how the betting exchange is able to use it. In addition, whenever the private key store is asked to sign a transaction and release funds, it requires the betting exchange to provide valid proof that the transaction to be signed is correct and authentic.

The best way to illustrate how the key store works is through an example. Imagine that Alice and Bob have bet against each other on the outcome of the *2015 Wimbledon final*, with Alice betting for *Roger Federer* and Bob betting for *Rafael Nadal*. Let's say that the outcome of the final was that *Roger Federer* was declared the winner and so Alice won the bet. Figure 29 shows the interactions between Alice, Bob, the betting exchange, the *Blockchain*, the private key store and the sports authority. In the diagram we can see that both Alice and Bob paid into separate and unique bet addresses for their bets on the final. As previously discussed, the betting exchange takes Alice and Bob's bet information and embeds it into the *Blockchain*. This allows them to prove that their payments were for their bets. When the event began, any unmatched portions of Alice and Bob's bets were returned to them. When the event finished the sports authority settled the bets by announcing that *Roger Federer* won the final and embedded this outcome into the *Blockchain*. The betting exchange then paid Alice her winnings.

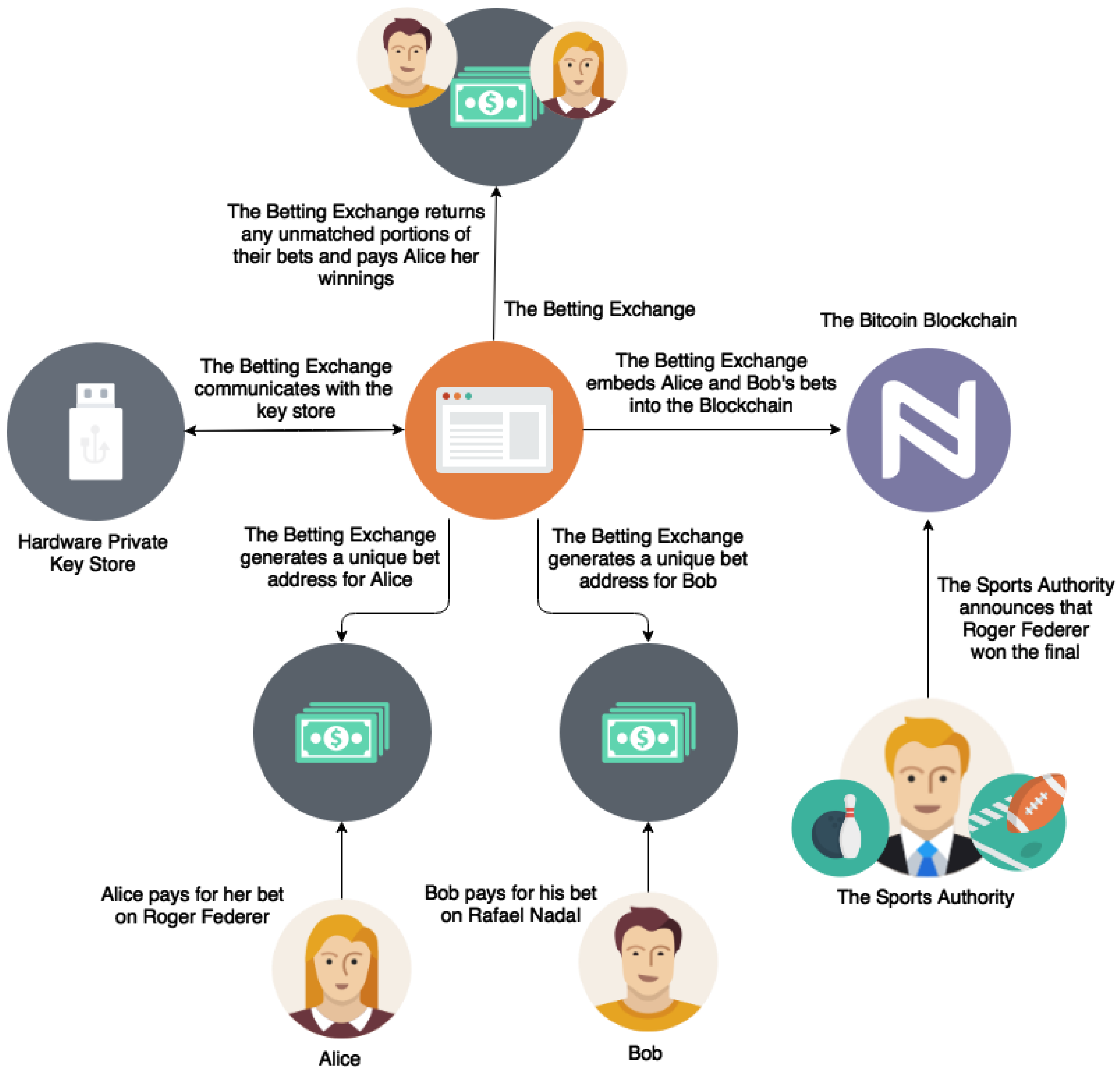

The Betting Exchange returns any unmatched portions of their bets and pays Alice her winnings
The Betting Exchange
The Bitcoin Blockchain
The Betting Exchange communicates with the key store
The Betting Exchange embeds Alice and Bob's bets into the Blockchain
Hardware Private Key Store
The Betting Exchange generates a unique bet address for Alice
The Betting Exchange generates a unique bet address for Bob
The Sports Authority announces that Roger Federer won the final
The Sports Authority
Alice pays for her bet on Roger Federer
Bob pays for his bet on Rafael Nadal
Alice
Bob

Figure 29: An example of Alice and Bob who bet on the *2015 Wimbledon final*. This diagram highlights the interactions between the various parties.

All throughout the example, the betting exchange has been communicating with the private key store. Looking again at the interactions we can see that there are effectively just 4 operations the betting exchange requires from the key store. These are, to *generate a new address*, to *create a bet proof*, to *return an unmatched portion* of a bet, and finally, to *pay the winning amount*. This means that the hardware device only needs to provide 3 interface functions:

1. The first function required from the key store is to generate a new bet address and a signed proof for that bet. When a user creates a new bet, the betting exchange contacts the key store and asks it to generate a new *Bitcoin* address for that bet, as well as to sign a bet proof that it can send to the *Bitcoin* network. Figure 30 shows how this function works. The betting exchange gives the key store the bet information when it makes a request for a new address. The information included in the request contains the event, the market,

the odds, and the outcome the user selected, as well as the user's *Bitcoin* address and a template transaction to sign. The key store then generates a new *Bitcoin* private key and address, uses that address along with the given information to calculate the bet hash, sets the *OP_RETURN* in the transaction and signs the bet proof for the exchange. The private key is encrypted before it is written to the device, using a public encryption key that has already been bootstrapped to the device. The nice thing to note here is that the key store does not need to share the private key with the exchange, but instead just returns the new bet address and the signed transaction that contains the bet proof. The new bet address is returned to the user and the signed transaction is finally broadcast to the network.

It is worth noting here that when the private key is written to the hardware device, the bet and user information it was given is also associated with that key. This has some useful safety properties we can draw upon in the next function.

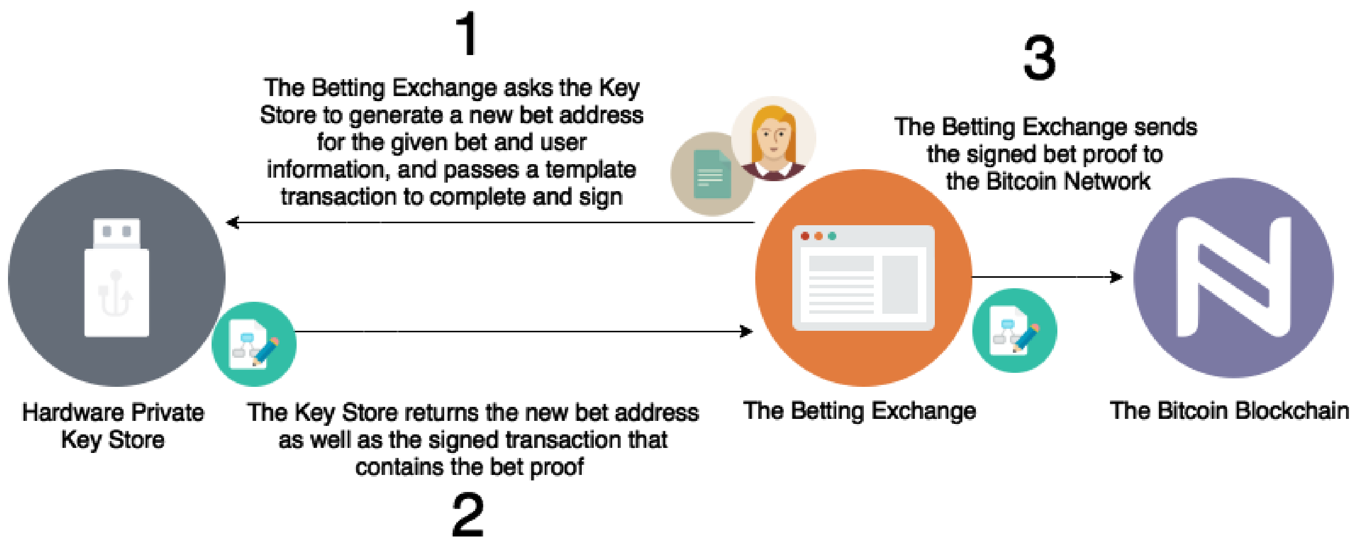

1
The Betting Exchange asks the Key Store to generate a new bet address for the given bet and user information, and passes a template transaction to complete and sign
3
The Betting Exchange sends the signed bet proof to the Bitcoin Network
Hardware Private Key Store
The Key Store returns the new bet address as well as the signed transaction that contains the bet proof
2
The Betting Exchange
The Bitcoin Blockchain

Figure 30: The betting exchange making a request for a new bet address and signed bet proof from the key store.

2. The second function that is required from the key store is to return an unmatched portion of a bet. Because bets can be both partially matched and unmatched, when an event begins the unmatched portion of a user's bet needs to be returned to them. This is because our solution does not support *in-play* betting and so once the event begins, no more bet matches can occur. Therefore, it makes sense to send the unmatched portion of a user's bet back to them when the event begins. This also reduces the amount of time we need to hold excess money. One of the nice properties about storing the bet information along with the private key in the hardware device is that when a request is made to return an unmatched portion of the user's bet, the device will only sign a transaction if the return address is the one that was originally specified by the user when the bet was created. This prevents an attacker from asking the wallet to return the amount to a different address.

   When this request is made, the amount to return is specified by the betting exchange. Figure 31 shows how this function works. The betting exchange passes the key store an unsigned transaction that returns this amount to the user. The key store then takes the requested amount and unsigned transaction and validates the transaction before signing it internally, in the device, and returning the signed transaction to the betting exchange. In order to prevent this function from being abused by an attacker (e.g. requesting the return of an amount greater than what was originally paid or is actually unmatched), we require 2 additional functions. The first function will tell the key store how much was paid into the bet address by the user, by passing in the raw signed transaction that was used to make their payment. The key store validates this transaction, and if it is correct, stores the amount received by the user and associates that amount with the bet address. Once this amount has been set, it cannot be changed.

The second additional function that we require is one that continuously updates the amount matched for a bet. Initially this amount is zero, but as other users decide to match this bet, the amount increases. We therefore keep the key store up to date with the amount currently matched for a bet. In order to secure this function, we can impose a strictly monotonic requirement for the values it is given. This means that as the amount matched is continuously updated by the exchange, the amount matched in the key store can never be set to a value lower than what it was previously at.

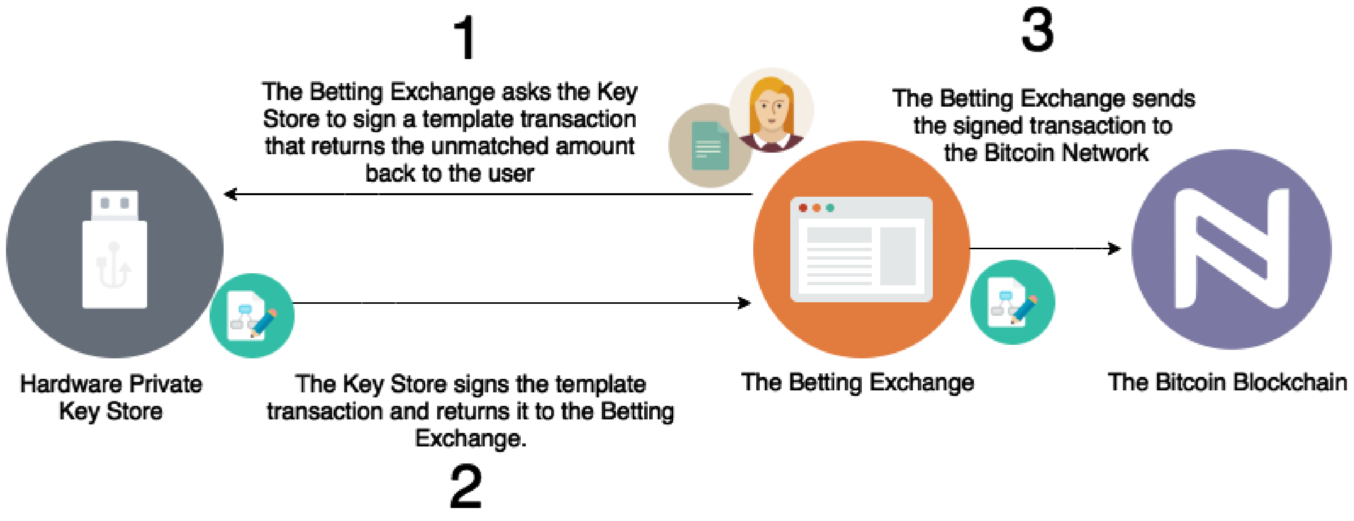

1
The Betting Exchange asks the Key Store to sign a template transaction that returns the unmatched amount back to the user
3
The Betting Exchange sends the signed transaction to the Bitcoin Network
Hardware Private Key Store
The Key Store signs the template transaction and returns it to the Betting Exchange.
2
The Betting Exchange
The Bitcoin Blockchain

Figure 31: The betting exchange making a request to return the unmatched portion of a user's bet.

Integrating these two additional functions into the interface of the key store allows us to prevent an attacker from requesting more than what was originally paid, and also prevents them from requesting more than is currently unmatched. In addition, because the key store signs the given transaction template internally, the private keys held inside it are never exposed outside the device. The only thing an attacker could therefore do to abuse this interface call, assuming they had access to the server, is to return the currently unmatched portion of a user's bet back to them. As previously mentioned, we can pass the decryption key to the key store when a return request is made. This is so that the key store can decrypt the appropriate key on the device and sign the transaction. When the transaction is signed, the key store writes over the decryption key and destroys it. Note that in an implementation of this, the communication between the betting exchange and the key store will be encrypted via a shared and bootstrapped encryption scheme. This is to prevent an attacker from reading any information sent between the two entities.

3. The final function that we require from the key store is to pay the winner. Rather than simply requesting the key store to pay a specific amount to some address, we can make use of the *bet* and *outcome proofs* that were embedded into the *Blockchain* earlier. Figure 32 shows how this function works. In our example, when the request to pay Alice's winnings is

made to the key store, we pass in the entire group of bets that were matched together (e.g. Alice and Bob's bets). We also pass in the *bet* and *outcome* transactions embedded into the *Blockchain*, the original data that hashes to the *OP_RETURNS* in those transactions, and finally an unsigned template transaction that pays Alice her winnings.

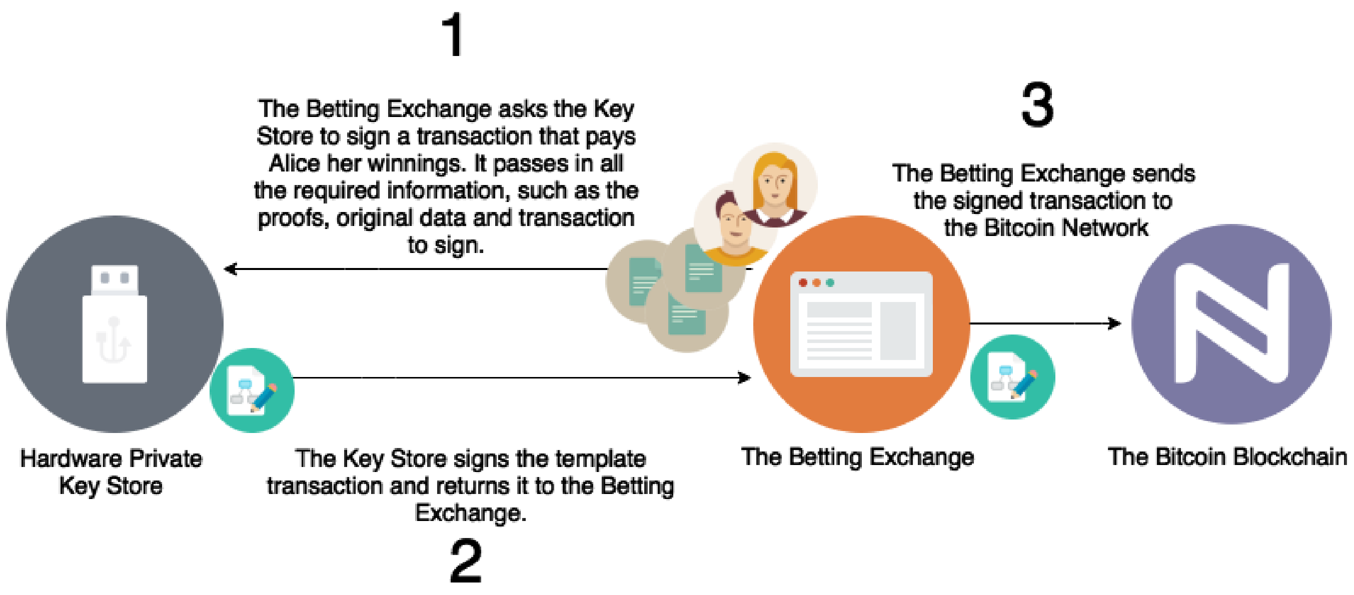

1
The Betting Exchange asks the Key Store to sign a transaction that pays Alice her winnings. It passes in all the required information, such as the proofs, original data and transaction to sign.
3
The Betting Exchange sends the signed transaction to the Bitcoin Network
Hardware Private Key Store
The Key Store signs the template transaction and returns it to the Betting Exchange.
2
The Betting Exchange
The Bitcoin Blockchain

Figure 32: The architecture of the *Bitcoin Betting Exchange*

The key store can then use all of this information to verify that the unsigned transaction is indeed the settlement of this bet group. It does so by checking firstly that the bets in the group are valid (e.g. that they were generated by the key store originally, that they have been paid for using a signed transaction from the user, that they can be matched appropriately for the specified odds, and that the amounts matched for those bets are consistent with the received payments). The key store can then check that the data for those bets hash correctly to the signed *bet proofs* it has been given. It can also check that the outcome proof it has been given is also correct (e.g. that it is an outcome for the same event and market as the bets and that it has been signed by the appropriate sports authority).

Having checked that the bets and outcome are all valid and for the same event, the key store can then check that the unsigned transaction it has been given by the exchange is for the appropriate address and amount. This can be done by looking up the odds of the original bet and the amount currently matched to determine the amount to pay (minus the fee from the betting exchange). It can then double check that this payment is going to the correct account by looking at the original user address given to it when the bet was generated. If all of these checks hold, the key store is then happy to sign the transaction locally and return the signed transaction back to the betting exchange. If not however, it refuses to sign the transaction.

Just like in the previous function that returns the unmatched bet amount to the user, we also pass the decryption key to the key store when a payment request is made. In addition, this request is also encrypted.

All in all, our hardware based, smart private key store device supports 5 interface functions, or methods, in total. These are:

1. Generating a new *Bitcoin* address and signed bet proof for a specific bet.
2. Updating the payment received for a specific bet. This is the payment sent by the user.
3. Updating the amount matched for a specific bet.
4. Returning the unmatched portion of a specific bet. This is the remaining amount of the user's payment not yet matched when the event began.
5. Paying the user for their winning bet.

The hardware device could make use of *hierarchical deterministic* generation of *Bitcoin* addresses when generating new addresses for users. The advantage of this is that in order to backup the keys held on the device, we would not need to repeatedly and manually copy the private keys off the device. Instead, we could just store the *seed* used to generate those addresses. Of course, this seed would need to be appropriately chosen to prevent weak-seed attacks and could be hard coded into the device under encryption to prevent it from being modified by an attacker. The backup of the seed would also need to be stored safely, preferably using a *paper wallet*, in a vault, under several layers of encryption. The decryption keys for that seed could then be fragmented and distributed amongst various parties to make it more difficult for an attacker to gain access to the original seed. *Bitcoin Trezor* also take this approach to address generation and backup the seeds that are used to deterministically generate new addresses. This prevents against hardware failure.

In addition, to protect the money that the betting exchange makes from each bet, when a bet group is settled, the portion that goes to the betting exchange could be transferred directly into a *cold wallet*. This *cold wallet*, just like the back-up of the seed, could be stored as a paper wallet and kept securely locked away under several layers of encryption.

4.2.5 Security Recommendations for Sports Authorities

As our solution also relies the security of the private keys held by the sports authorities, we recommend that careful consideration is given to the way in which their keys are protected. Exposing these keys to an attacker could not only be disastrous for them, but also allow an attacker to announce incorrect event outcomes and hence have devastating consequences for us. In order to prevent this, significant attention needs to be paid to the way in which these keys are secured. For example, they could use *Bitcoin Trezor* to announce event outcomes. The operator responsible for announcing these outcomes can be given the hash that they need to embed by our website. They can then use *Bitcoin Trezor* to embed these outcomes into the *Blockchain* by connecting it to a computer, signing the transaction in the hardware, and then disconnecting and safely storing the device until another event outcome needs to be made known. As discussed, *Bitcoin Trezor* never exposes the keys to the outside environment, which is ideal in this scenario. Physical access to the *Bitcoin Trezor* device also needs to be monitored.

5. The Bitcoin Betting Exchange

In this chapter we introduce the *Bitcoin Betting Exchange*. We discuss how it works from a user perspective, showcasing the web application's front-end design and run through several examples. We also look at the way in which sports authorities are able to interact with our platform to perform bet settlement and announce event outcomes. Finally, we explore how bets and outcomes are verified and how the results of those verifications are presented to the user.

5.1 Introduction

The *Bitcoin Betting Exchange* is an anonymous betting exchange that allows users to place bets on various sports events using *Bitcoin*. As previously mentioned, in the background, section 2.2, betting exchanges match bets across users and take a small commission from the winner. Users are able to *back* an outcome in a specific market, choosing to place a bet for that outcome to occur. Likewise, users are also able to *lay* a specific outcome, choosing to place a bet that says the outcome does not occur. These two types of bet can therefore be matched against each other, assuming appropriate odds, and the bets settled when the outcome is determined.

The *Bitcoin Betting Exchange* is fully operational and currently running online at:
`https://bitcoin-betting.herokuapp.com`.

Likewise, a software implementation of the smart, private key store is also fully operational and currently running at:
`https://bitcoin-betting-wallet.herokuapp.com`.

Figure 33 shows a screen-shot of the home page displayed to users:

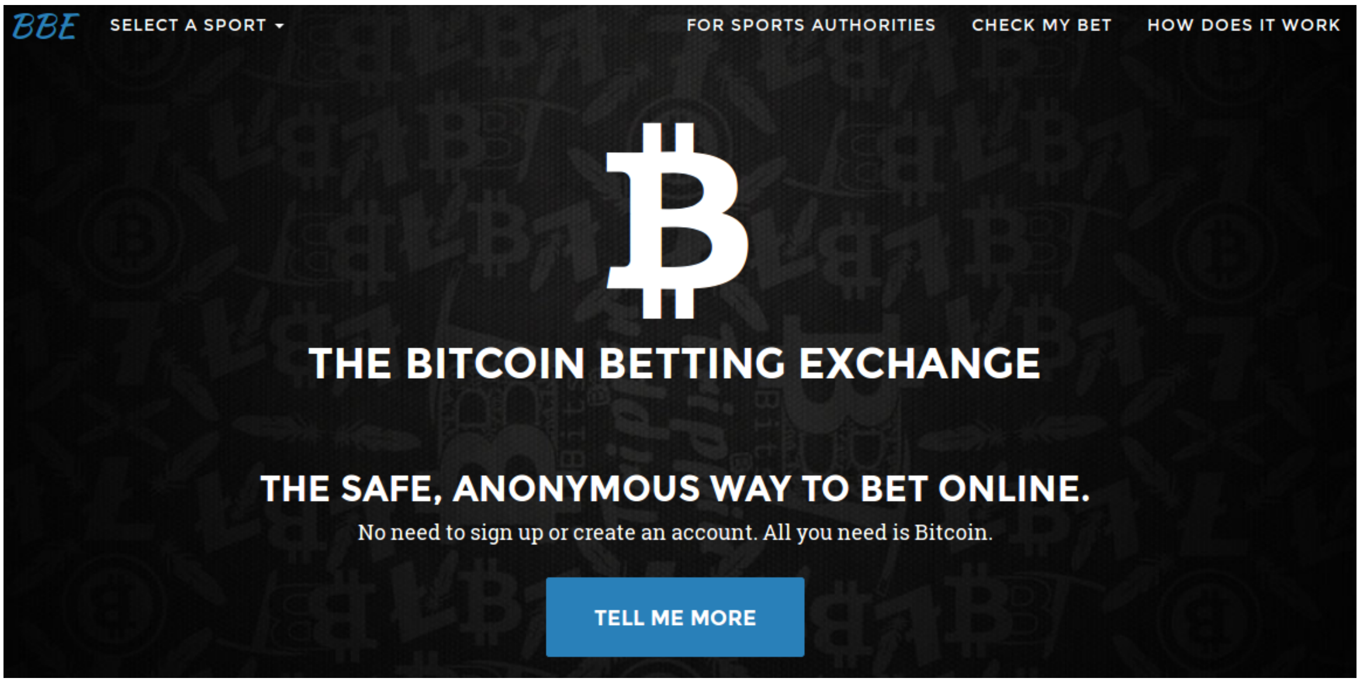
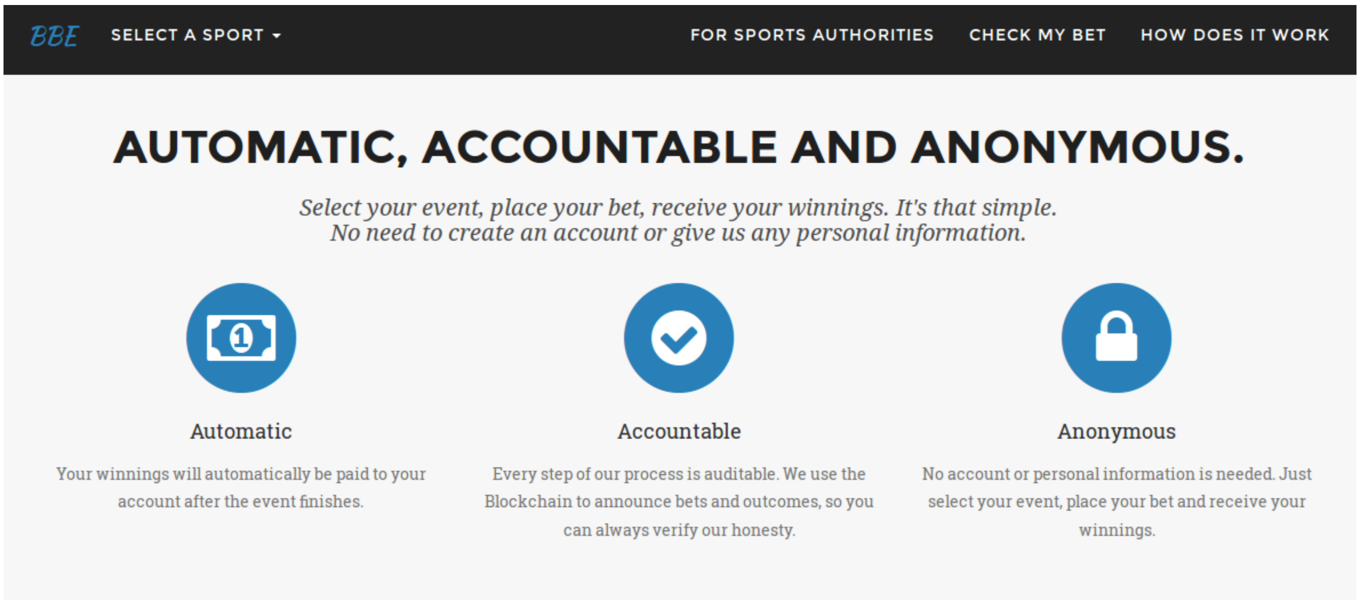

BBE
SELECT A SPORT
FOR SPORTS AUTHORITIES
CHECK MY BET
HOW DOES IT WORK
THE BITCOIN BETTING EXCHANGE
THE SAFE, ANONYMOUS WAY TO BET ONLINE.
No need to sign up or create an account. All you need is Bitcoin.
TELL ME MORE

Figure 33: The Bitcoin Betting Exchange Homepage

In order to make it easy for users to understand exactly how our application works and why they might want to use it, the very first option available to them is to view a summary of what our solution provides. This is accessible by clicking on the *Tell Me More* button, or by simply scrolling down the page. The main selling points that we provide are that our platform is *automatic* (settling bets and paying out winnings immediately), it is *accountable* (giving them concrete proof of any bets they place and the outcome of those bets) and it is *anonymous* (requiring no personal information or user sign up). We also provide brief instructions on how to get started. Figure 34 shows a screen-shot of this information.



AUTOMATIC, ACCOUNTABLE AND ANONYMOUS.

Select your event, place your bet, receive your winnings. It's that simple.
No need to create an account or give us any personal information.

Automatic

Your winnings will automatically be paid to your account after the event finishes.

Accountable

Every step of our process is auditable. We use the Blockchain to announce bets and outcomes, so you can always verify our honesty.

Anonymous

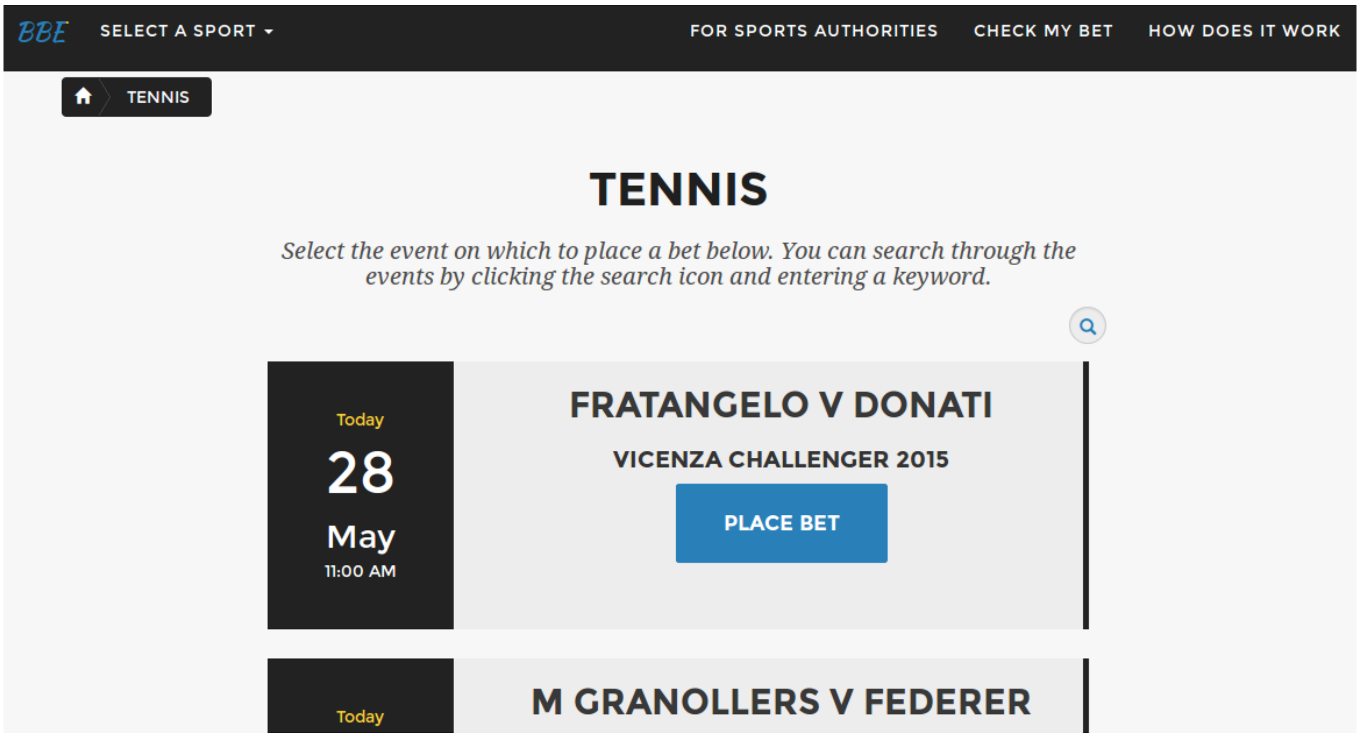
No account or personal information is needed. Just select your event, place your bet and receive your winnings.

Figure 34: Automatic, Accountable and Anonymous

The floating navigation bar at the top of the page provides users with quick access to the most important links, such as *Check My Bet* and *How Does it Work*. Furthermore the dropdown menu in the navigation bar provides direct access to the sports that users can choose to bet on.

When a sport is selected users are shown the current events for that sport. For convenience, these events are ordered by start time, meaning that the events shown first are the ones closest to starting (and the ones earliest to finishing). Figure 35 shows a screen-shot of the events available for *Tennis*. In addition, all events are searchable by keyword, giving users the ability to quickly access the events they wish to bet on. Searchable keywords include the title of the event, the competition the event is in, the players taking part and the date of the event. Note that because we do not support *in-play betting*, bets can only be placed on an event before the event has begun.

For convenience we only display events starting within the next 3-7 days to prevent an excessive number of events being displayed at once. There is an exception that large and popular events further in the future are also displayed. This allows users to bet on current events as well as gives them early notice of their favourite events in the future. Likewise if there are not many events occurring within the near future, we choose to display the earliest events for that sport regardless of whether or not they fall within the 3-7 day window. Note that the starting time for each event is displayed using the *time zone* of the users location, meaning that these times are relative to each user depending on where they are accessing the site from.



TENNIS

Select the event on which to place a bet below. You can search through the events by clicking the search icon and entering a keyword.

Today 28 May 11:00 AM

FRATANGELO V DONATI

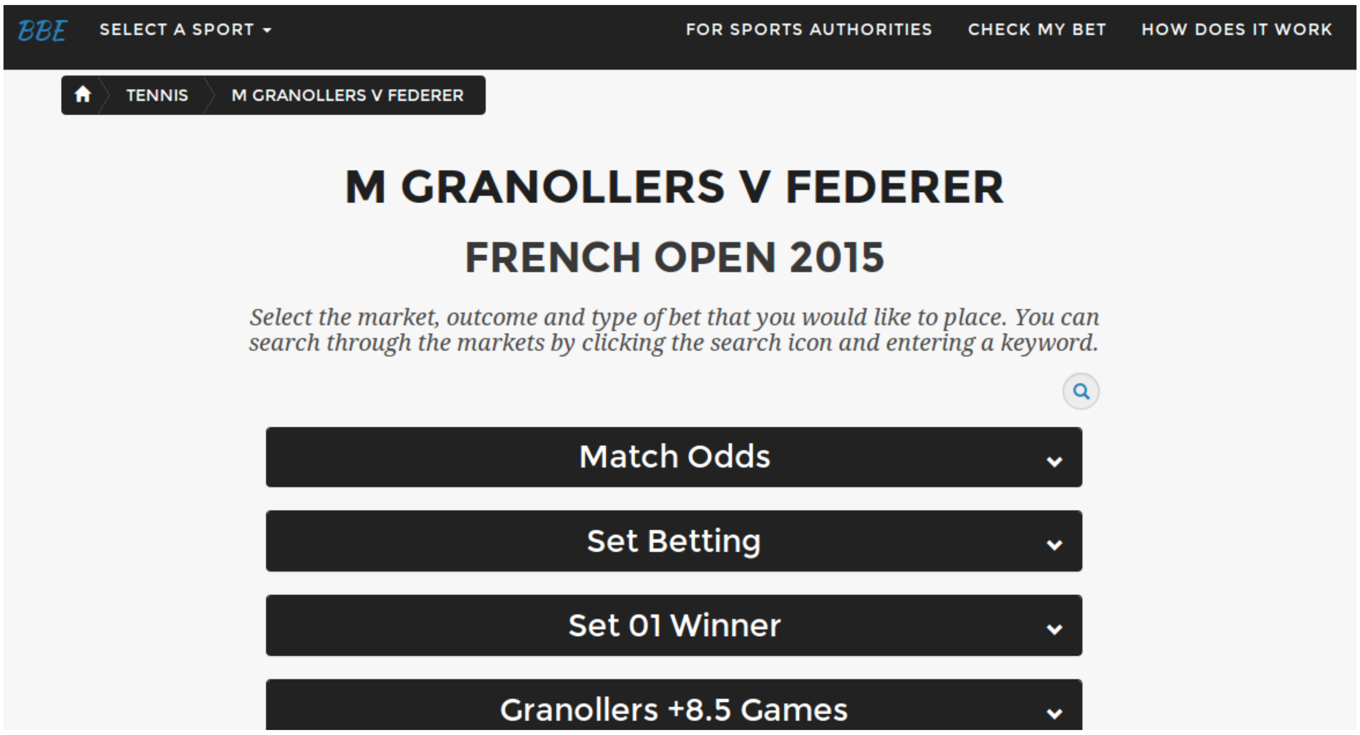
VICENZA CHALLENGER 2015

Today

M GRANOLLERS V FEDERER

Figure 35: The Tennis Events Displayed

When a user has decided the event on which they wish to bet, they can select it and will be shown the markets (types of bets) available for that event. Figure 36 shows a screen-shot of the various markets available for a tennis event.



M GRANOLLERS V FEDERER

FRENCH OPEN 2015

Select the market, outcome and type of bet that you would like to place. You can search through the markets by clicking the search icon and entering a keyword.

Figure 36: The Markets for a Tennis Event

Within each market users can see the currently available or unmatched bets placed by other users. This allows them to easily match a bet already offered, or to propose their own by entering their own odds. They also have access to a help button that displays additional information about the market, such as how the market works, and its rules and regulations. Figure 37 shows a screen-shot of the currently unmatched bets available for a specific market. Unmatched bets are displayed using a format that is common to many existing betting exchanges, that is, to display the bet odds and amount that are currently available for the user. Note that by default we use *decimal odds*. For example, in figure 37, there are currently 250,000 *satoshi* available to *back Phillip Kohlschreiber* at odds of *1.4*. This means that someone has already *layed Phillip* at the appropriate odds and amount to offer you the chance to *back* him.

Likewise, there are current 40,000 *satoshi* available to *lay Phillip* at *back* odds of 1.5. Note here that *lay* bets are always presented using the *back* odds of the bets that were already placed by other users. We do not show the *lay* odds available in this view. This might seem confusing to someone who has not used a betting exchange before. Odds are always shown in *back* format because it makes it easier to view the difference between what odds people are willing to back at, and what odds people are willing to lay at. For instance, if you were to choose to *lay Phillip* in this example by selecting the button displaying odds of 1.5, your lay odds would actually be 3.0. *If this is unclear, it may be helpful to re-read the background on betting exchanges and to see the relationship between back and lay odds.* In addition, if there are currently no unmatched bets available, users can choose to *back* or *lay* at their own proposed odds.

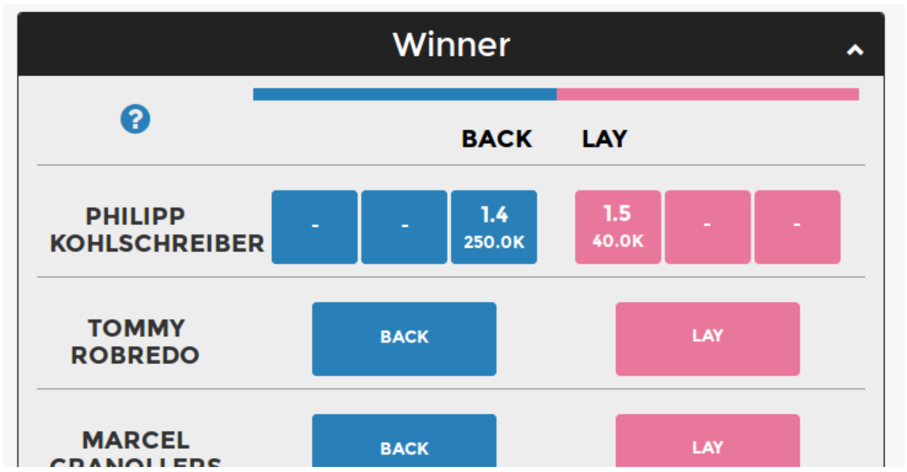
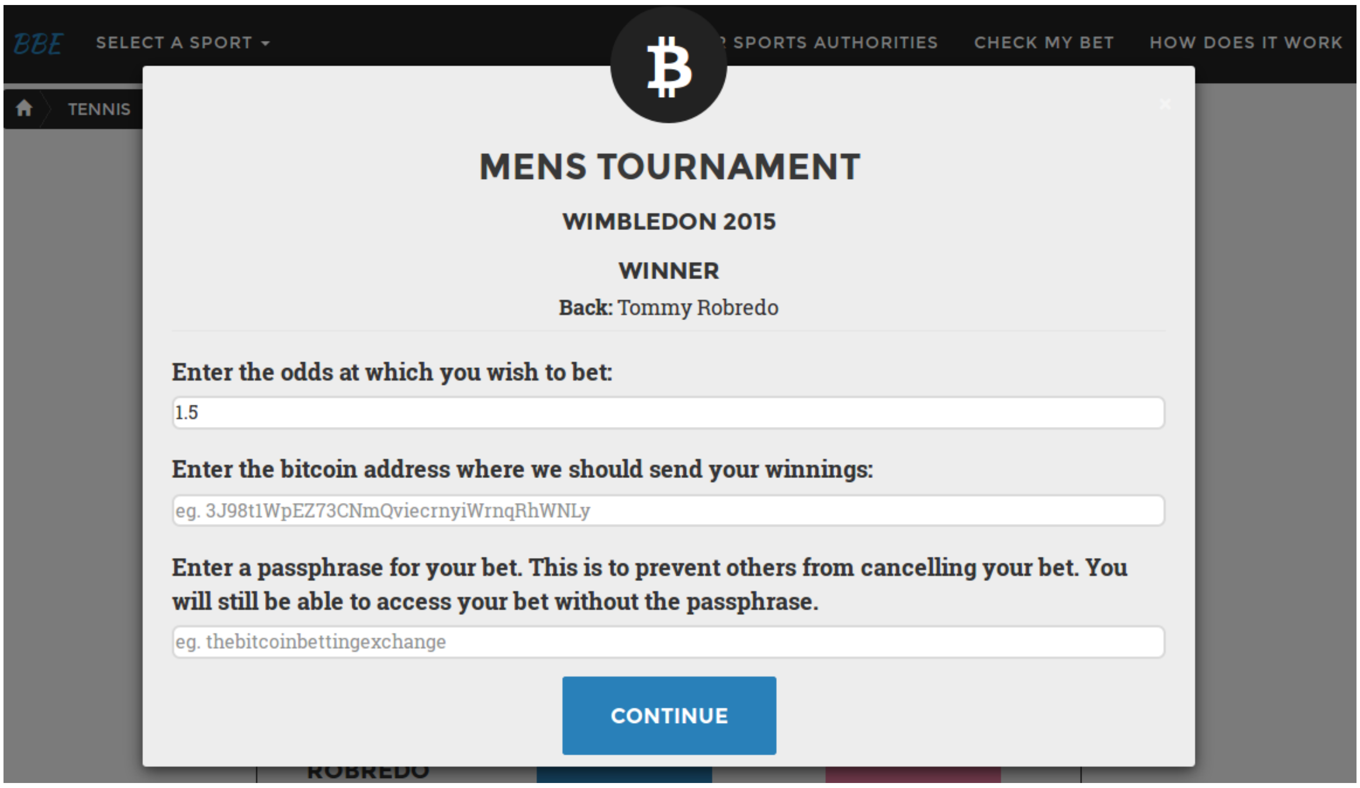


Figure 37: The Open Bets for a Tennis Market. The Market is *Winner*: “Who will win?”

Depending on whether or not users choose to match an existing bet, or propose their own, they can create a bet by clicking on one of these buttons and completing a bet form. To do this, they enter their own *Bitcoin* address (the address where their winnings will be sent) and a pass-phrase for their bet (to prevent others from being able to cancel it). No other information is required. Figure 38 shows a screen-shot of a user creating a back bet.



MENS TOURNAMENT

WIMBLEDON 2015

WINNER

Back: Tommy Robredo

Enter the odds at which you wish to bet:

1.5

Enter the bitcoin address where we should send your winnings:

eg. 3J98t1WpEZ73CNmQviecrnyiWrnqRhWNLy

Enter a passphrase for your bet. This is to prevent others from cancelling your bet. You will still be able to access your bet without the passphrase.

eg. thebitcoinbettingexchange

CONTINUE

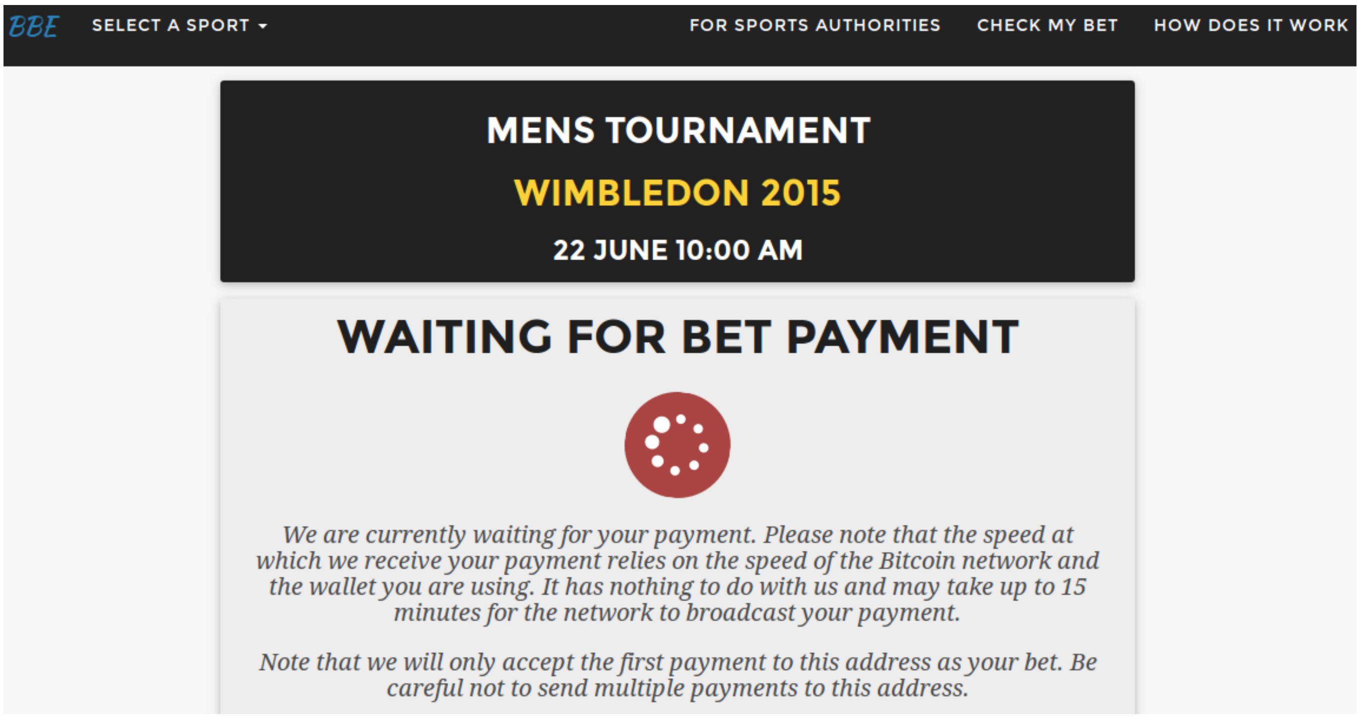
Figure 38: A user creating a *back* bet for a specific market and outcome.

When a bet is created, a *Bitcoin* address is generated that represents the bet for the user. This bet address is unique to each bet and is ultimately where the user will send their *Bitcoin* to. When the user creates a bet they will automatically be taken to the bet page, where they are able to monitor and interact with their bet. Figures 39 and 40 show screen-shots of this page for a bet that has just been created.

This page provides the user with all of the information that they might need to know regarding the current status of their bet. As we can see in the screen-shots below, the current status of the created bet is *Waiting For Bet Payment*, telling the user that we are still waiting to receive their payment. Each bet page highlights the bet address that the user needs to pay into and provides a scannable *QRcode* for use with mobile *Bitcoin* wallets.

When the user has paid into this address, the bet automatically updates with the amount received, without having to refresh the page. If their bet can be matched, or partially matched, with an existing bet, the two bets are updated and the status shown. Figures 41 and 42 show screen-shots of a bet that has been paid for and is completely matched. Note that users cannot see know who their bet was matched with.

Looking at figure 42 we can see that the amount received for the bet was 15,000 satoshi. Likewise, we were able to match that amount with another bet, so the total matched amount is also 15,000 satoshi, making the bet a complete match. Next to the payment received we also show an approximate value for the amount in *British Pounds*. This value is calculated using the exchange rate and so may change as the exchange rate fluctuates. Note that as is typical of betting exchanges, users are unable to cancel their bets once they have been matched or partially matched.



MENS TOURNAMENT

WIMBLEDON 2015

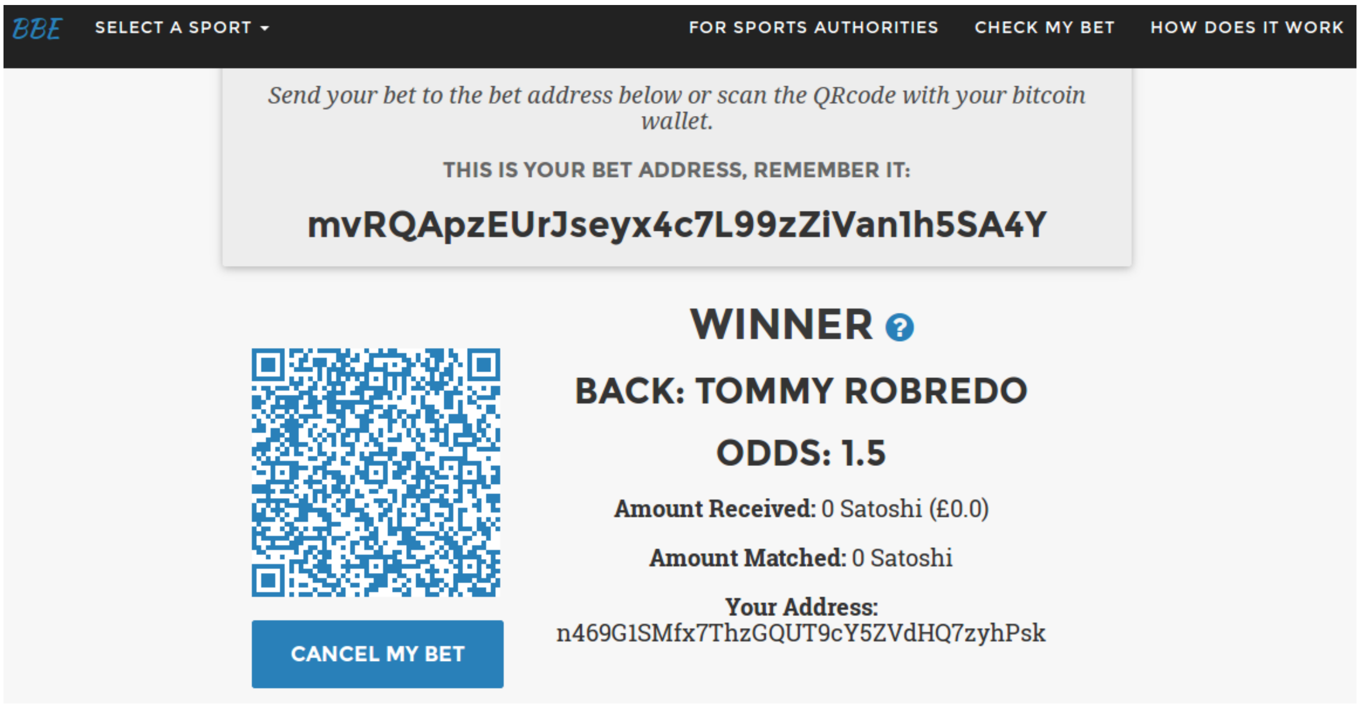
22 JUNE 10:00 AM

WAITING FOR BET PAYMENT

We are currently waiting for your payment. Please note that the speed at which we receive your payment relies on the speed of the Bitcoin network and the wallet you are using. It has nothing to do with us and may take up to 15 minutes for the network to broadcast your payment.

Note that we will only accept the first payment to this address as your bet. Be careful not to send multiple payments to this address.

Figure 39: The bet page for an unpaid bet. Screen-shot 1.



Send your bet to the bet address below or scan the QRcode with your bitcoin wallet.

THIS IS YOUR BET ADDRESS, REMEMBER IT:

mvRQApzEUrJseyx4c7L99zZiVan1h5SA4Y

WINNER

BACK: TOMMY ROBREDO

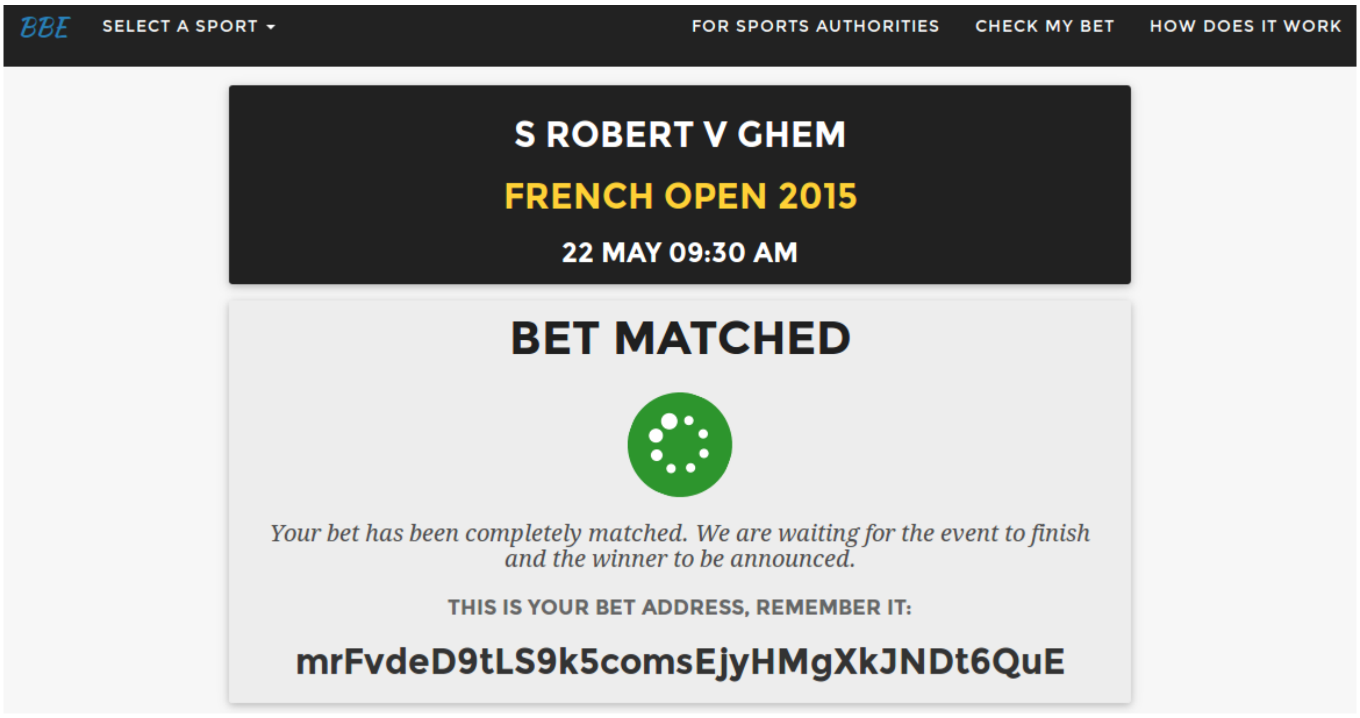
ODDS: 1.5

Amount Received: 0 Satoshi (£0.0)

Amount Matched: 0 Satoshi

Your Address:
n469G1SMfx7ThzGQUT9cY5ZVdHQ7zyhPsk

Figure 40: The bet page for an unpaid bet. Screen-shot 2.



S ROBERT V GHEM

FRENCH OPEN 2015

22 MAY 09:30 AM

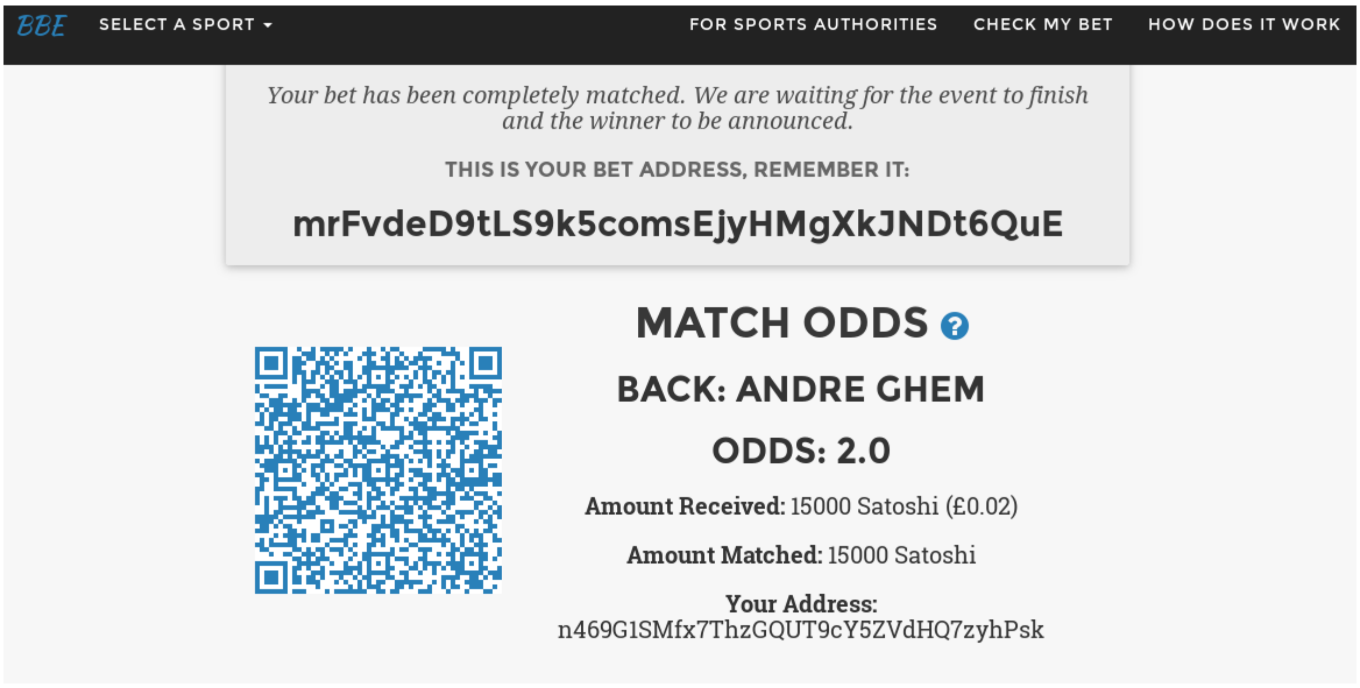
BET MATCHED

Your bet has been completely matched. We are waiting for the event to finish and the winner to be announced.

THIS IS YOUR BET ADDRESS, REMEMBER IT:

mrFvdeD9tLS9k5comsEjyHMgXkJNDt6QuE

Figure 41: The bet page for a matched bet. Screen-shot 1.



Your bet has been completely matched. We are waiting for the event to finish and the winner to be announced.

THIS IS YOUR BET ADDRESS, REMEMBER IT:

mrFvdeD9tLS9k5comsEjyHMgXkJNDt6QuE

MATCH ODDS

BACK: ANDRE GHEM

ODDS: 2.0

Amount Received: 15000 Satoshi (£0.02)

Amount Matched: 15000 Satoshi

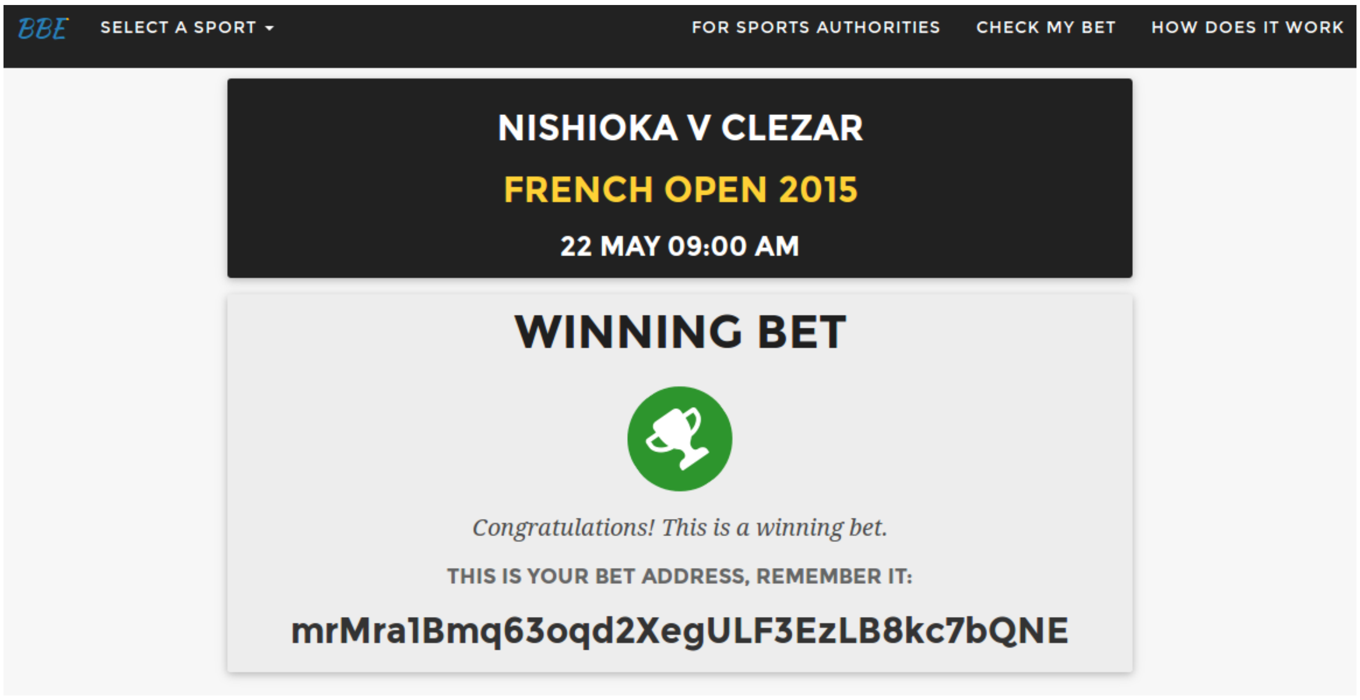
Your Address:
n469G1SMfx7ThzGQUT9cY5ZVdHQ7zyhPsk

Figure 42: The bet page for a matched bet. Screen-shot 2.

At this stage, when a bet has been matched or partially matched, the next step in the process is to wait for the event to finish and the result of the market to be announced. If a bet has only been partially matched, it can continue to be increasingly matched as time goes on, all the way up until the event begins. At this stage markets are closed and the winnings paid out only once the event has finished and the outcome made known.

Some bets may contain portions that are unmatched when the event begins, for example, if a bet has received no matches, or if a bet is only partially matched. In these cases the unmatched portions of the bets are returned to the user at the start of the event.

Figures 43 and 44 show screen-shots of a winning bet. If you look at figure 44 you can see a summary of the bet. The bet was placed at odds of 2.0, 60,000 satoshi was paid by the user, only 40,000 was matched and 70,000 was won. If we perform a quick calculation, we can see that 20,000 satoshi should have been returned when the event started (as this was the unmatched portion of the bet). The bet was made at odds of 2.0, meaning the total winnings of the matched portion should be $40,000 \times 2.0$, equalling 80,000 satoshi. The user only received 70,000. This might seem strange at first, but if you remember back to the background section of *Bitcoin*, the *Bitcoin* network requires 10,000 satoshi as a miner fee, making up the missing portion of the winnings. Note that in this example, we, the betting exchange, have taken zero commission on this bet.



NISHIOKA V CLEZAR

FRENCH OPEN 2015

22 MAY 09:00 AM

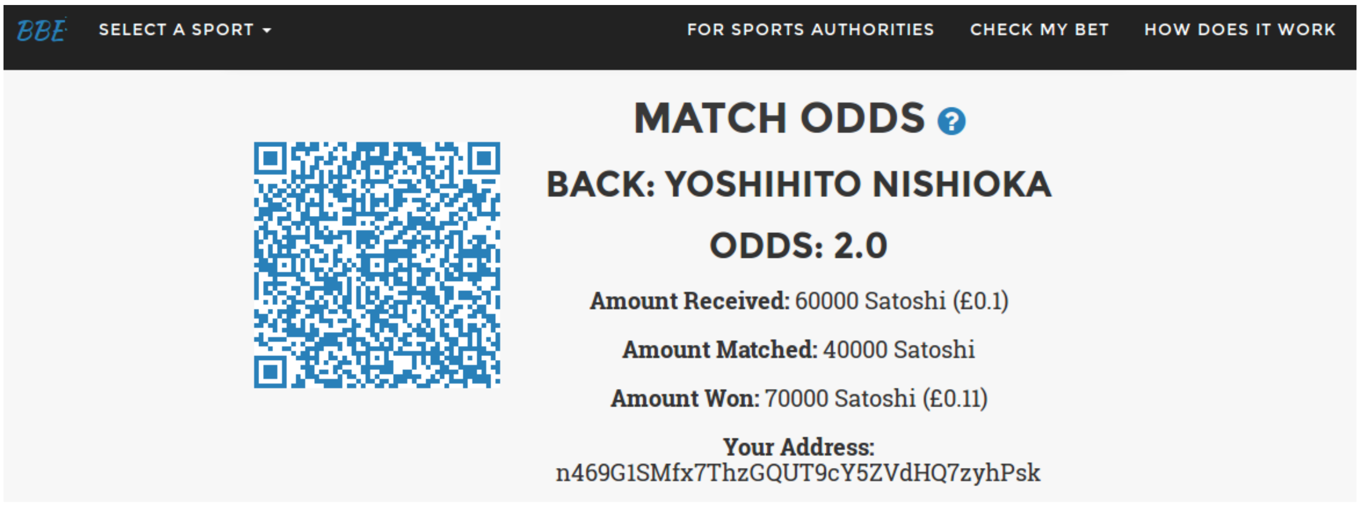
WINNING BET

Congratulations! This is a winning bet.

THIS IS YOUR BET ADDRESS, REMEMBER IT:

mrMra1Bmq63oqd2XegULF3EzLB8kc7bQNE

Figure 43: The bet page for a winning bet. Screen-shot 1.



MATCH ODDS

BACK: YOSHIHITO NISHIOKA

ODDS: 2.0

Amount Received: 60000 Satoshi (£0.1)

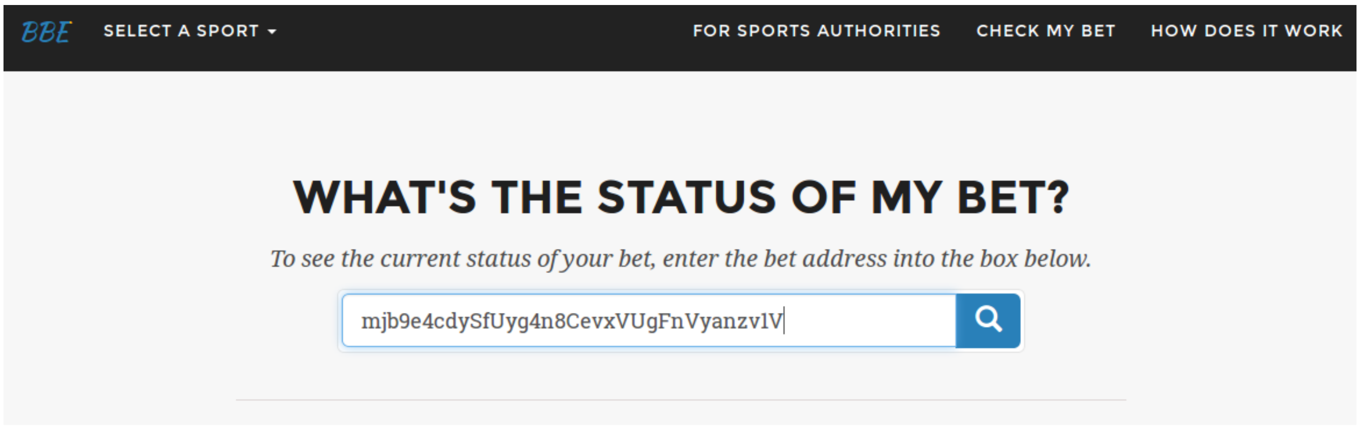
Amount Matched: 40000 Satoshi

Amount Won: 70000 Satoshi (£0.11)

Your Address:
n469G1SMfx7ThzGQUT9cY5ZVdHQ7zyhPsk

Figure 44: The bet page for a winning bet. Screen-shot 2.

Users are always able to look-up their bet status using the *Check My Bet* page. This can be done by entering the *Bitcoin* address corresponding to their bet into a search box on that page. If the user loses their bet address, they can always find it again by looking up the transaction in the *Bitcoin* wallet they used to make the payment. Figure 45 shows a screen-shot of this page. Entering the bet address of the bet into the search box directs the user back to their bet page.



WHAT'S THE STATUS OF MY BET?

To see the current status of your bet, enter the bet address into the box below.

mjb9e4cdySfUyg4n8CevxVUgFnVyanzv1V

Figure 45: The bet page for a winning bet. Screen-shot 2.

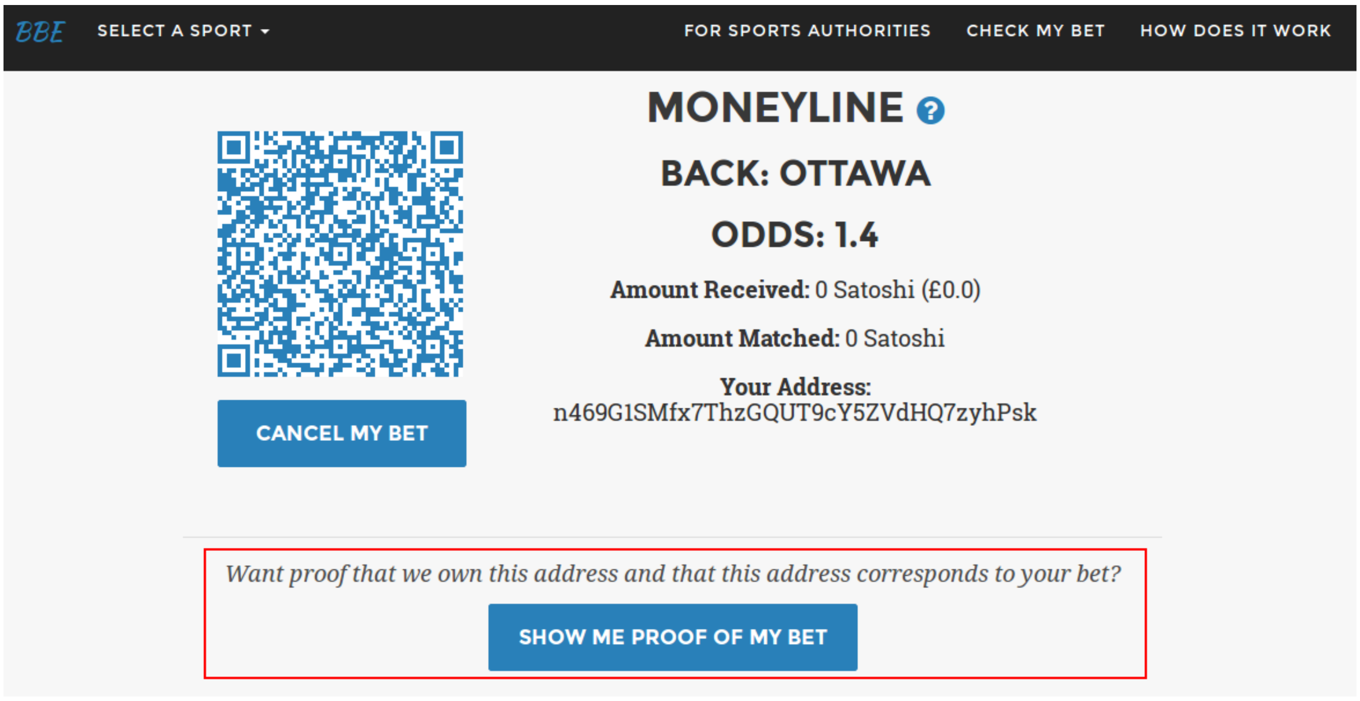
5.2 Proof of Bet

In the previous section, we saw a user create a bet, pay for it, win that bet, and receive their winnings. But, what if the betting exchange decided not to pay the user's winnings, and instead pocketed the money, removing all traces of the bet from the website. One might argue that the user could still prove their payment, because the transaction exists publicly in the *Blockchain*. Although this is true, they wouldn't be able to say that this transaction was for their bet. This is because the transaction only shows a payment from one address to another address, and the destination address of that transaction could be owned by anyone. The user has no way to prove that the destination address of their transaction was for their specific bet.

What makes the *Bitcoin Betting Exchange* unique is that despite the anonymity it provides, it also gives users a concrete way to verify and prove that the bet address they have paid into is owned by the betting exchange. What's more, they can not only prove ownership of that address, but also that the address represents their exact bet (i.e. the event, the market, the outcome and the odds they selected). This proof is provided before any payment to the address is required, allowing users to verify this information before they make their payment. Furthermore, this proof can also be used as concrete evidence in the event that a dispute occurs between the betting exchange and the bettor.

As discussed in section 4.1.6 we make use of the *OP_RETURN* transaction type to publicly announce and store bet proofs whenever a new bet is created. The *Bitcoin Betting Exchange* makes it easy for users to check these proofs and verify that these hashes exist in the *Blockchain* before making a payment into their bet address. It does so by providing a link on the bottom of every bet page that gives users direct access to the proof. Figure 46 shows a screen-shot of an unpaid bet and a link, highlighted in red, that allows users to verify their bet before paying.

When a user clicks on the *"Show Me Proof of My Bet"* button at the bottom of the page, they are taken to the *Bet Announcement* page. The *Bet Announcement* page contains all the information a user needs to verify that their bet and data has been hashed correctly and announced (embedded) into the *Blockchain*. This page contains step by step instructions on how this bet proof works, as well as explanations on how to go about verifying it. It also gives users the option to download a pdf of the proof to keep for their own records, so they never have to worry about losing the original (unhashed) data. Figures 47, 48 and 49 show screen-shots of the *Bet Announcement* page.



MONEYLINE

BACK: OTTAWA

ODDS: 1.4

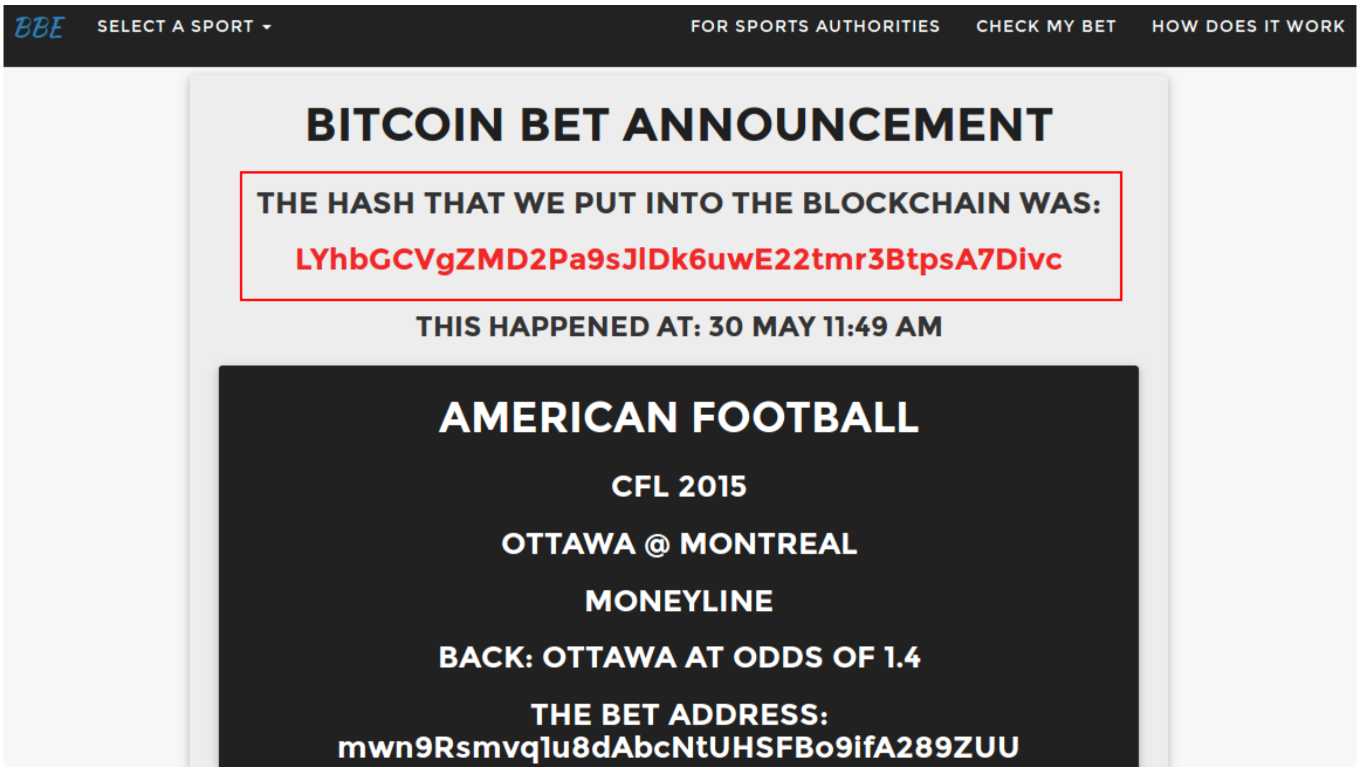
Amount Received: 0 Satoshi (£0.0)

Amount Matched: 0 Satoshi

Your Address:
n469G1SMfx7ThzGQUT9cY5ZVdHQ7zyhPsk

Want proof that we own this address and that this address corresponds to your bet?

Figure 46: The link on a bet page that takes users to their bet proof.



BITCOIN BET ANNOUNCEMENT

THE HASH THAT WE PUT INTO THE BLOCKCHAIN WAS:
LYhbGCVgZMD2Pa9sJlDk6uwE22tmr3BtpsA7Divc

THIS HAPPENED AT: 30 MAY 11:49 AM

AMERICAN FOOTBALL

CFL 2015

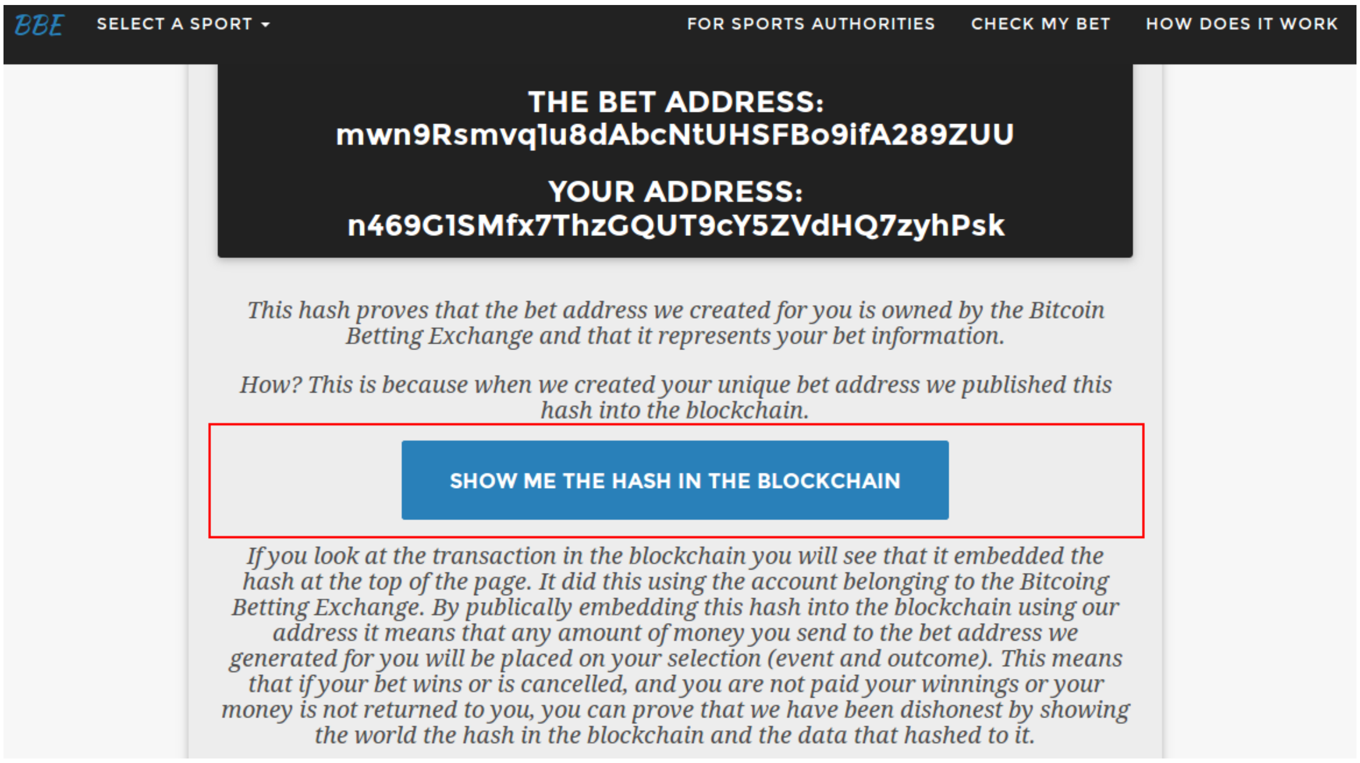
OTTAWA @ MONTREAL

MONEYLINE

BACK: OTTAWA AT ODDS OF 1.4

THE BET ADDRESS:
mwn9Rsmvq1u8dAbcNtUHSFBo9ifA289ZUU

Figure 47: The Bet Announcement Page. Screen-shot 1.



THE BET ADDRESS:
mwn9Rsmvq1u8dAbcNtUHSFBo9ifA289ZUU

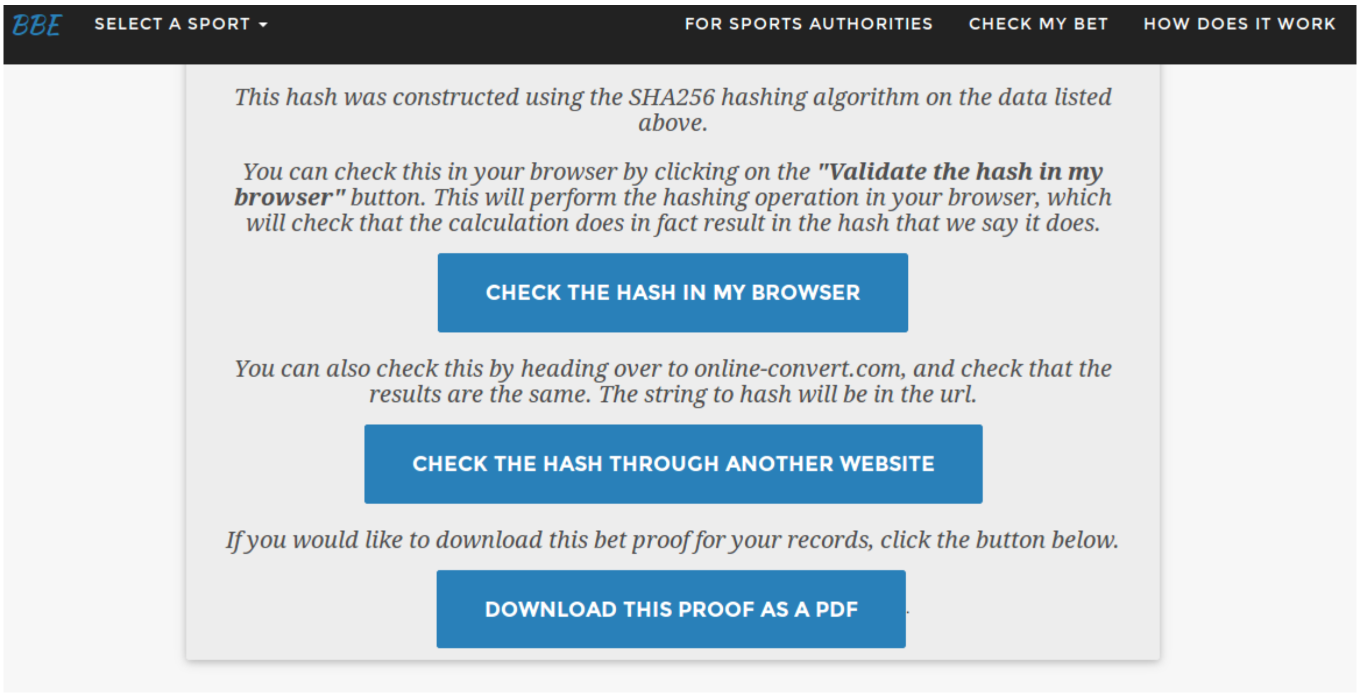
YOUR ADDRESS:
n469G1SMfx7ThzGQUT9cY5ZVdHQ7zyhPsk

This hash proves that the bet address we created for you is owned by the Bitcoin Betting Exchange and that it represents your bet information.

How? This is because when we created your unique bet address we published this hash into the blockchain.

If you look at the transaction in the blockchain you will see that it embedded the hash at the top of the page. It did this using the account belonging to the Bitcoing Betting Exchange. By publically embedding this hash into the blockchain using our address it means that any amount of money you send to the bet address we generated for you will be placed on your selection (event and outcome). This means that if your bet wins or is cancelled, and you are not paid your winnings or your money is not returned to you, you can prove that we have been dishonest by showing the world the hash in the blockchain and the data that hashed to it.

Figure 48: The Bet Announcement Page. Screen-shot 2.



This hash was constructed using the SHA256 hashing algorithm on the data listed above.

*You can check this in your browser by clicking on the **"Validate the hash in my browser"** button. This will perform the hashing operation in your browser, which will check that the calculation does in fact result in the hash that we say it does.*

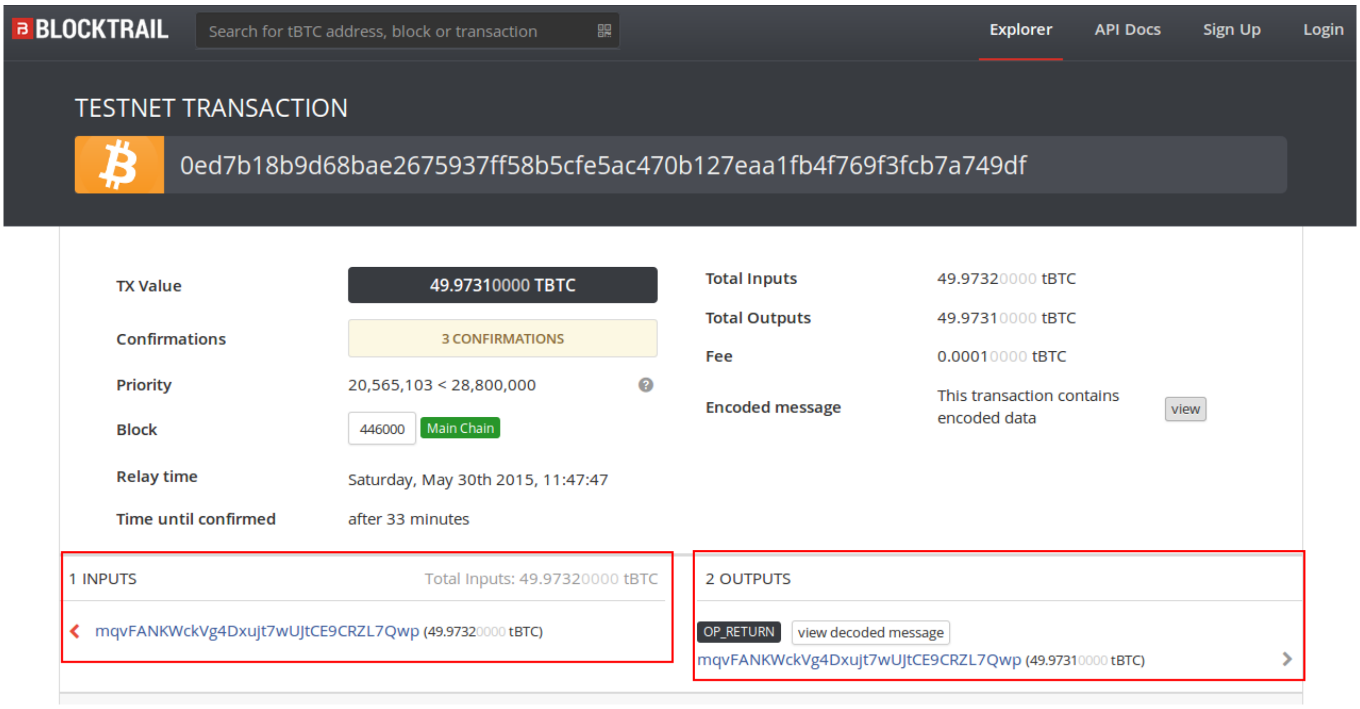
You can also check this by heading over to online-convert.com, and check that the results are the same. The string to hash will be in the url.

If you would like to download this bet proof for your records, click the button below.

Figure 49: The Bet Announcement Page. Screen-shot 3.

If we look at figures 47 and 48 we can see that the *Bitcoin Betting Exchange* embedded the hash `LYhbGCVgZMD2Pa9sJlDk6uwE22tmr3BtpsA7Divc` into the *Blockchain* on the 31st May at 11:49 AM (highlighted in red). Underneath the hash (in the black box) a summary of the original data is displayed. This is the data that was hashed and represents the unique bet. As well as an explanation on what this information represents, we also provide a direct link to the transaction in the *Blockchain*, so that users can see the *OP_RETURN* transaction for themselves and verify that it embeds the correct hash.

Clicking on *"Show Me The Hash In The Blockchain"*, highlighted in red, figure 48, forwards the user to *Blocktrail.com* [27], a third party website. *Blocktrail.com* is a Bitcoin transaction explorer that allows users to manually view and explore raw blocks and transactions in the *Blockchain*. Figure 50 shows a screen-shot of the bet *OP_RETURN* transaction on *Blocktrail.com*. By looking at this screen-shot we can see that this transaction contains a single input, highlighted in red, and two outputs, also highlighted in red. The input belongs to the *Bitcoin* address `mqvFANKWckVg4Dxujt7wUJtCE9CRZL7Qwp`, which is the known public address of the *Bitcoin Betting Exchange*. The first output is an *OP_RETURN* transaction embedding a "decoded message", the second transaction sends the remainder of the spent input back to the *Bitcoin Betting Exchange*. If a user clicks "view decoded message" they will see the value of the string embedded into the *OP_RETURN* transaction. Figure 51 shows what is displayed. Highlighted in red is the alpha-numeric (*base64*) representation of the *OP_RETURN* that was embedded. This shows the string `LYhbGCVgZMD2Pa9sJlDk6uwE22tmr3BtpsA7Divc`, which is the same string that represents the bet. Note that *Blocktrail.com* choose to prepend an open bracket to the beginning of the their *OP_RETURN* strings.


BLOCKTRAIL

Search for tBTC address, block or transaction

Explorer API Docs Sign Up Login

TESTNET TRANSACTION

0ed7b18b9d68bae2675937ff58b5cfe5ac470b127eaa1fb4f769f3fcb7a749df

TX Value	49.97310000 TBTC	Total Inputs	49.97320000 tBTC
Confirmations	3 CONFIRMATIONS	Total Outputs	49.97310000 tBTC
Priority	20,565,103 < 28,800,000	Fee	0.00010000 tBTC
Block	446000 Main Chain	Encoded message	This transaction contains encoded data view
Relay time	Saturday, May 30th 2015, 11:47:47		
Time until confirmed	after 33 minutes		

1 INPUTS Total Inputs: 49.97320000 tBTC

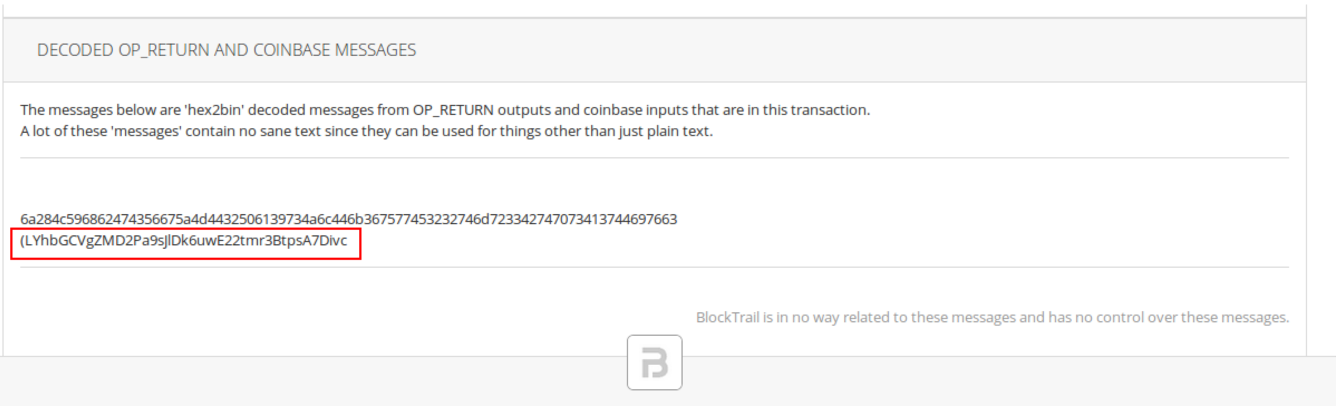
mqvFANKWckVg4Dxujt7wUJtCE9CRZL7Qwp (49.97320000 tBTC)

2 OUTPUTS

OP_RETURN view decoded message

mqvFANKWckVg4Dxujt7wUJtCE9CRZL7Qwp (49.97310000 tBTC)

Figure 50: The transaction that embedded the bet hash into the Blockchain, as viewed by *Blocktrail.com*


DECODED OP_RETURN AND COINBASE MESSAGES

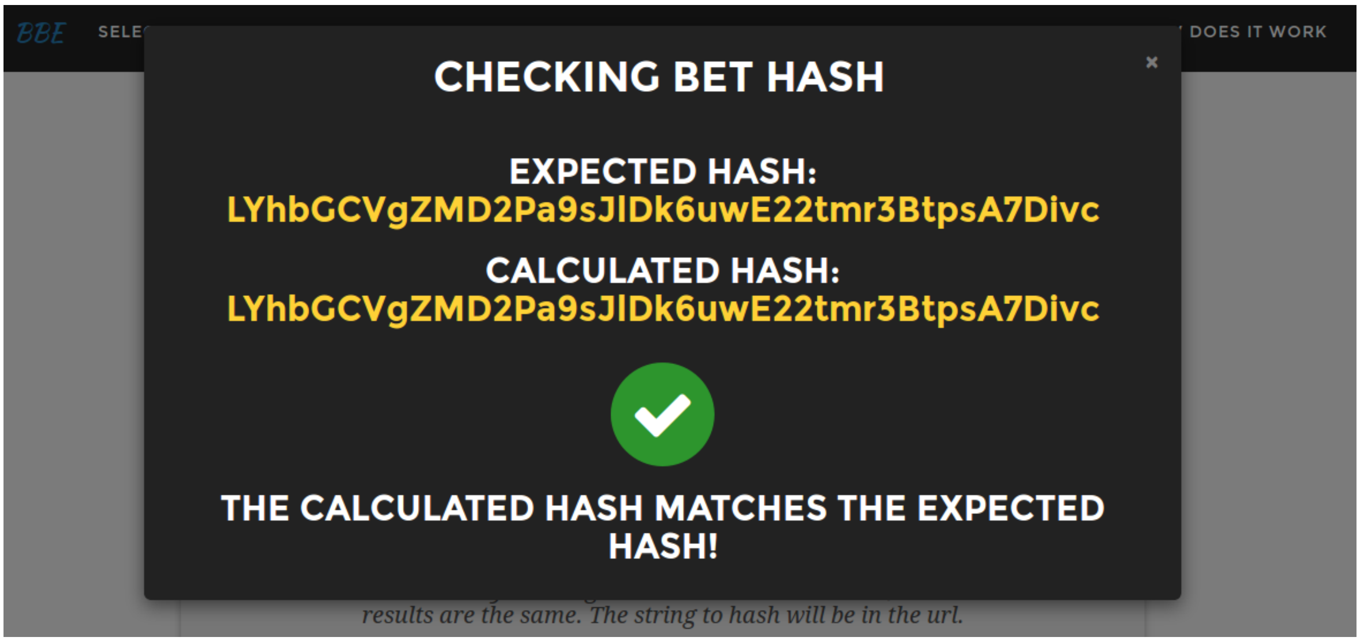
The messages below are 'hex2bin' decoded messages from OP_RETURN outputs and coinbase inputs that are in this transaction.
A lot of these 'messages' contain no sane text since they can be used for things other than just plain text.

```
6a284c596862474356675a4d4432506139734a6c446b3675774532322746d7233427470734137446976663
(LYhbGCVgZMD2Pa9sJlDk6uwE22tmr3BtpsA7Divc
```

BlockTrail is in no way related to these messages and has no control over these messages.

Figure 51: The decoded message embedded into the OP_RETURN transaction.

Having manually verified that the hash has been embedded into the *Blockchain* by the *Bitcoin Betting Exchange*, the next step for the user is to verify that the bet data does in fact hash to the string that was embedded. The *Bet Announcement* page, figure 49, explains that the hashing algorithm used to produce this hash is *SHA256*, the very same hashing algorithm used predominantly throughout *Bitcoin* (see section 2.1). The *Bet Announcement* page provides two ways for users to verify that the data does in-fact hash to the same result. The first way is performed directly in the browser when the user clicks the *“Check the Hash in My Browser”* button, figure 49. On the client side the data is hashed using a javascript implementation of *SHA256*, and the result is compared to the hash that was embedded in the *Blockchain*. A pop-up box displays the result to the user, showing the expected hash, the calculated hash and the result of the comparison. Figure 52 shows this result.



CHECKING BET HASH

EXPECTED HASH:
LYhbGCVgZMD2Pa9sJlDk6uwE22tmr3BtpsA7Divc

CALCULATED HASH:
LYhbGCVgZMD2Pa9sJlDk6uwE22tmr3BtpsA7Divc

THE CALCULATED HASH MATCHES THE EXPECTED HASH!

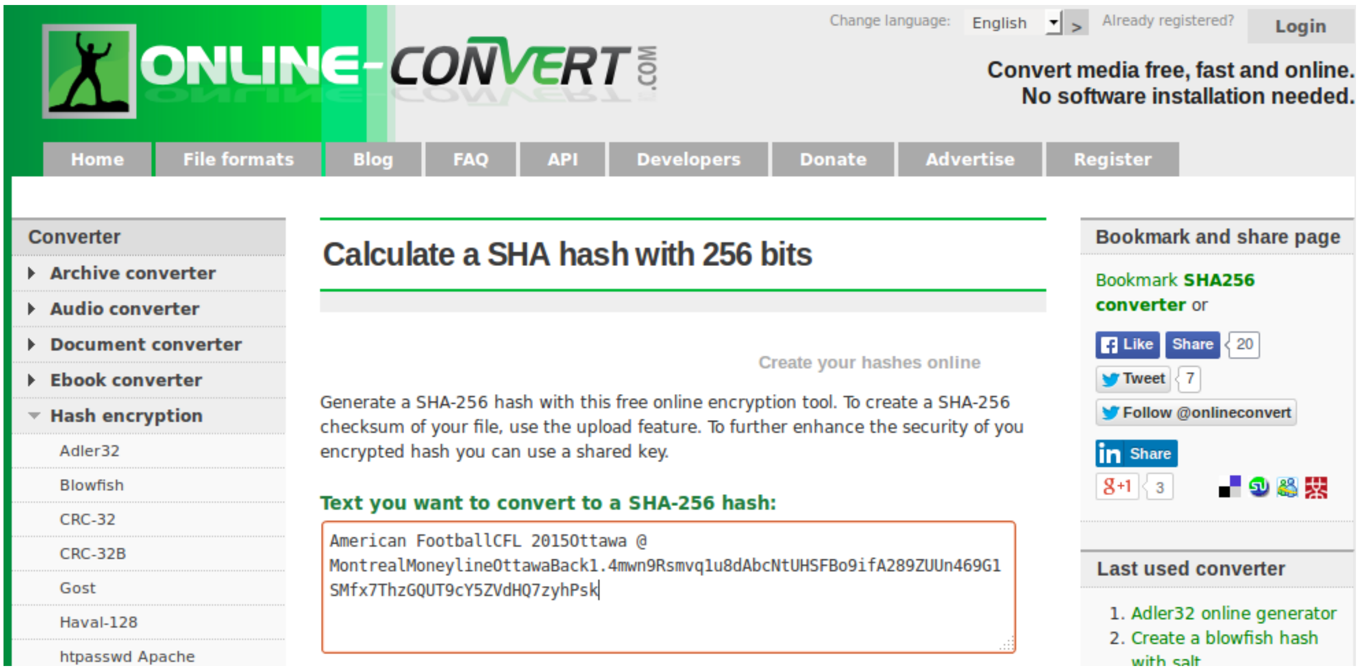
results are the same. The string to hash will be in the url.

Figure 52: The result of the client-side SHA256 hashing operation on the bet data.

The second way for a user to verify the hash is through a third party website. In order to prevent the *Bitcoin Betting Exchange* from just claiming that the hashes are the same, we also provide a way for users to perform the hash outside of our application. To do this, users can select the *"Check The Hash Through Another Website"* button, figure 49, and they will be forwarded to *online-convert.com* [90], a website that calculates various encryption, hashing and data conversion operations. The data to hash is passed in directly as a query parameter to the url, and so the user can enter this string into the website and calculate the *SHA256* hash. Figure 53 shows the user passing the string into a text-box on *online-convert.com* (highlighted in red). Figure 54 shows the result. As can be seen in figure 54 the *base64* representation of the result, also highlighted in red, is `LYhbGCVgZMD2Pa9sJlDk6uwE22tmr3BtpsA7DivczYI=`, the first 40 characters of which match the embedded hash. Note here that we only embed the first 40 characters of any hash due to the previous limitation that existed on the *OP_RETURN* transaction type.

Having analysed the *OP_RETURN* transaction in the *Blockchain* manually and verified that the data does correctly hash to the embedded hash, we provide the user with the ability to download this bet proof as a pdf. This allows them to always have access to this proof, should they ever need to use it in the case of a dispute. It also means that should the *Bitcoin Betting Exchange* ever try to remove this proof from the website in an attempt to misbehave, users will still be able to prove their bets. We chose to allow users to download the pdf instead of having the proof sent to them automatically via email because, as previously mentioned, requiring email addresses from users compromises their anonymity. Figure 55 shows a downloadable bet proof in pdf format. The pdf includes all of the same steps and instructions as the web page.

Similarly to how users can look up a bet using their bet address on the *"Check My Bet"* page, users can also look up a bet proof using the hash that was embedded into the *Blockchain*. This provides quick access to a bet proof without needing to go via the bet page. Figure 56 shows a user searching for their bet proof. Note that in order to look up a bet proof via this page, the user needs to enter the pass-phrase for the bet. This is important because it prevents an attacker from being able to abuse the system. For example, an attacker might collect all the *OP_RETURNS* published into the *Blockchain* by our public address. They could then take these *OP_RETURNS* and look them up using the *"Check My Bet"* page. This would give them access to the bet proofs and also bet pages, meaning they could perform two types of attack. Firstly, if the bet was unmatched, it would allow the attacker to cancel the bet without the users knowledge or consent. Although the user wouldn't lose any money as their bet would simply be returned to them, this would be inconvenient and annoying. Secondly, it would allow an attacker to monitor exactly what people bet on (e.g. by seeing which *Bitcoin* addresses make payments to these bet addresses). Requiring the user to enter their pass-phrase in order to look up a bet proof prevents this. Note that users can still access their bet and hence bet proof using their unique bet address without the pass-phrase. In this case we rely on the uniqueness of the bet address to prevent an attacker from being able to get unauthorised access to this information. Figure 57 shows a user entering their bet pass-phrase.



Convert media free, fast and online.
No software installation needed.

Calculate a SHA hash with 256 bits

Create your hashes online

Generate a SHA-256 hash with this free online encryption tool. To create a SHA-256 checksum of your file, use the upload feature. To further enhance the security of you encrypted hash you can use a shared key.

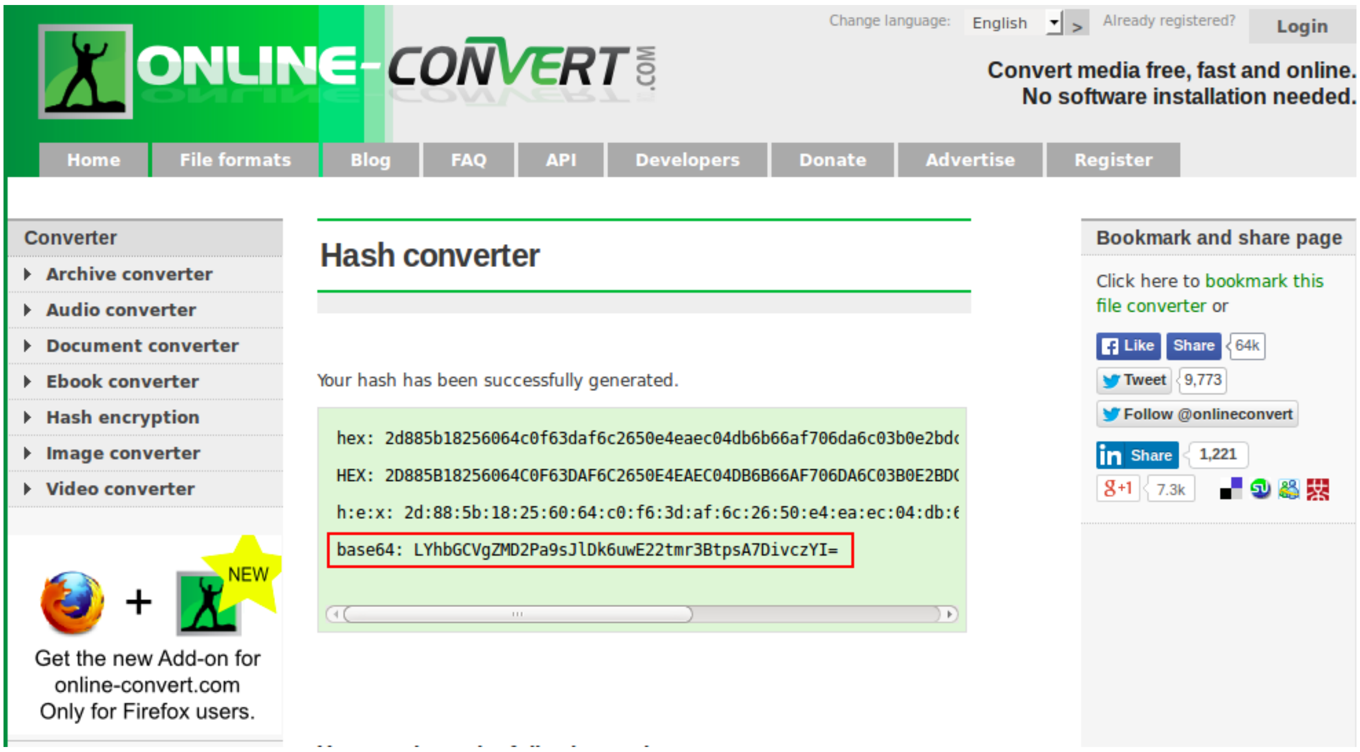
Text you want to convert to a SHA-256 hash:

```
American FootballCFL 2015Ottawa @
MontrealMoneylineOttawaBack1.4mwn9Rsmvq1u8dAbcNtUHSFBo9ifA289ZUUn469G1
SMfx7ThzGQUT9cY5ZVdHQ7zyhPsk
```

Bookmark SHA256 converter or

1. Adler32 online generator
2. Create a blowfish hash with salt

Figure 53: Using the *online-convert.com* tool to hash the bet data.

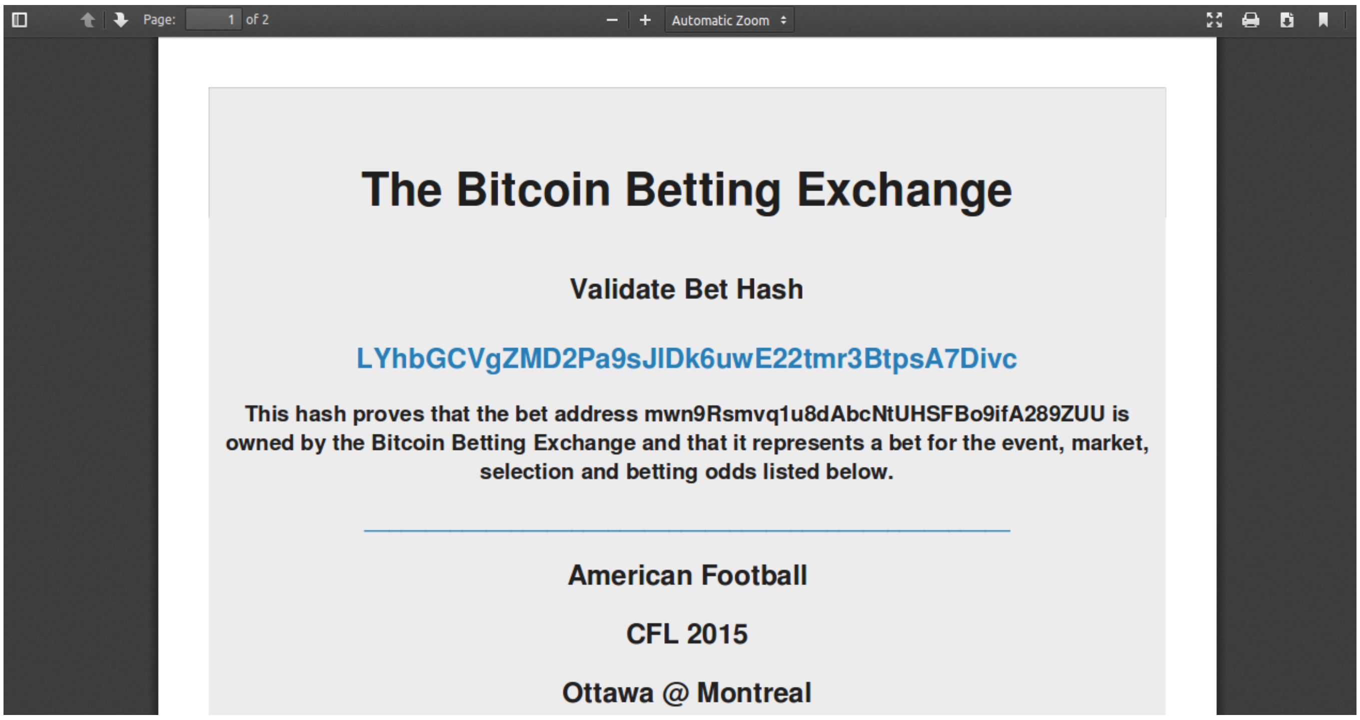


Hash converter

Your hash has been successfully generated.

```
hex: 2d885b18256064c0f63daf6c2650e4eaec04db6b66af706da6c03b0e2bd
HEX: 2D885B18256064C0F63DAF6C2650E4EAEC04DB6B66AF706DA6C03B0E2BD
h:e:x: 2d:88:5b:18:25:60:64:c0:f6:3d:af:6c:26:50:e4:ea:ec:04:db:
base64: LYhbGCVgZMD2Pa9sJlDk6uwE22tmr3BtpsA7DivczYI=
```

Figure 54: The resulting hash shown by *online-convert.com*



The Bitcoin Betting Exchange

Validate Bet Hash

LYhbGCVgZMD2Pa9sJlDk6uwE22tmr3BtpsA7Divc

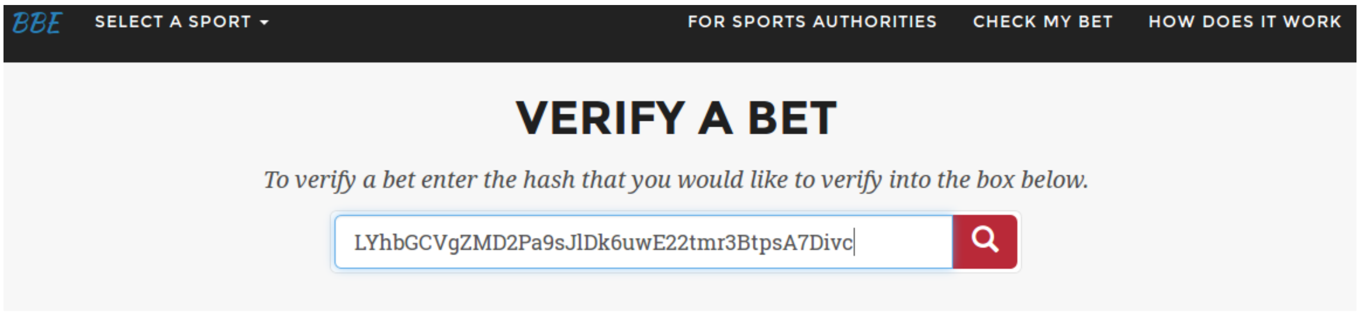
This hash proves that the bet address mwn9Rsmvq1u8dAbcNtUHSFBo9ifA289ZUU is owned by the Bitcoin Betting Exchange and that it represents a bet for the event, market, selection and betting odds listed below.

American Football

CFL 2015

Ottawa @ Montreal

Figure 55: The pdf download available for a bet proof.

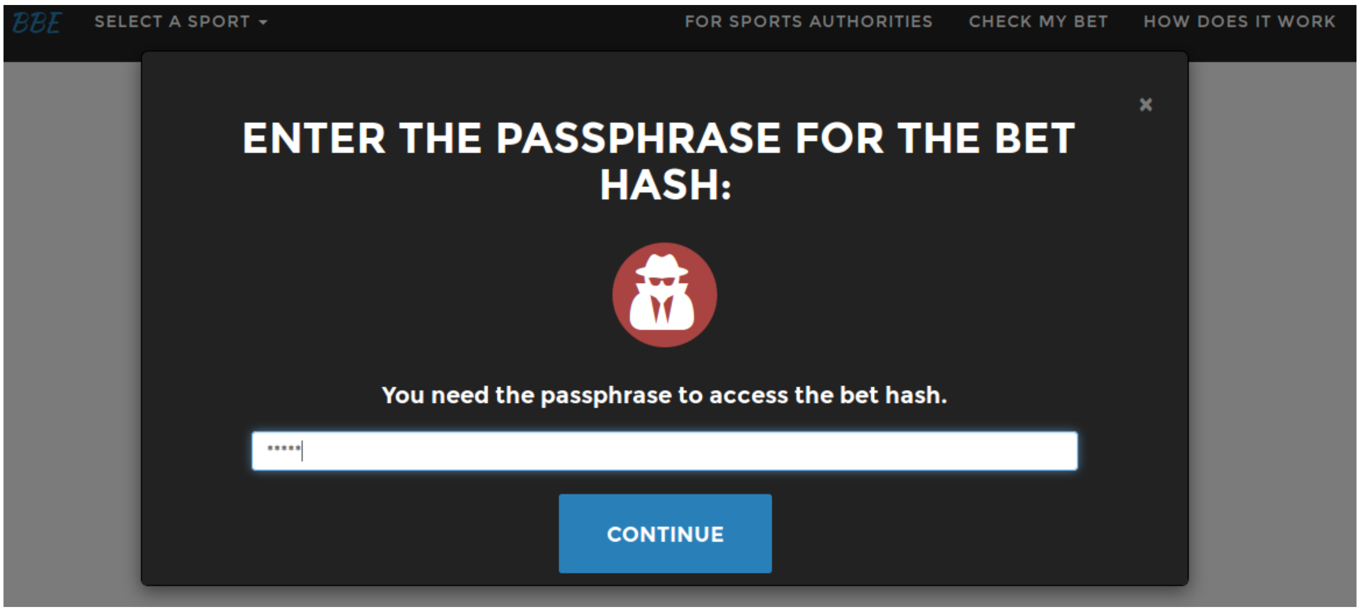


VERIFY A BET

To verify a bet enter the hash that you would like to verify into the box below.

LYhbGCVgZMD2Pa9sJlDk6uwE22tmr3BtpsA7Divc

Figure 56: A user searching for their bet proof via the *"Check My Bet"* page.



ENTER THE PASSPHRASE FOR THE BET HASH:

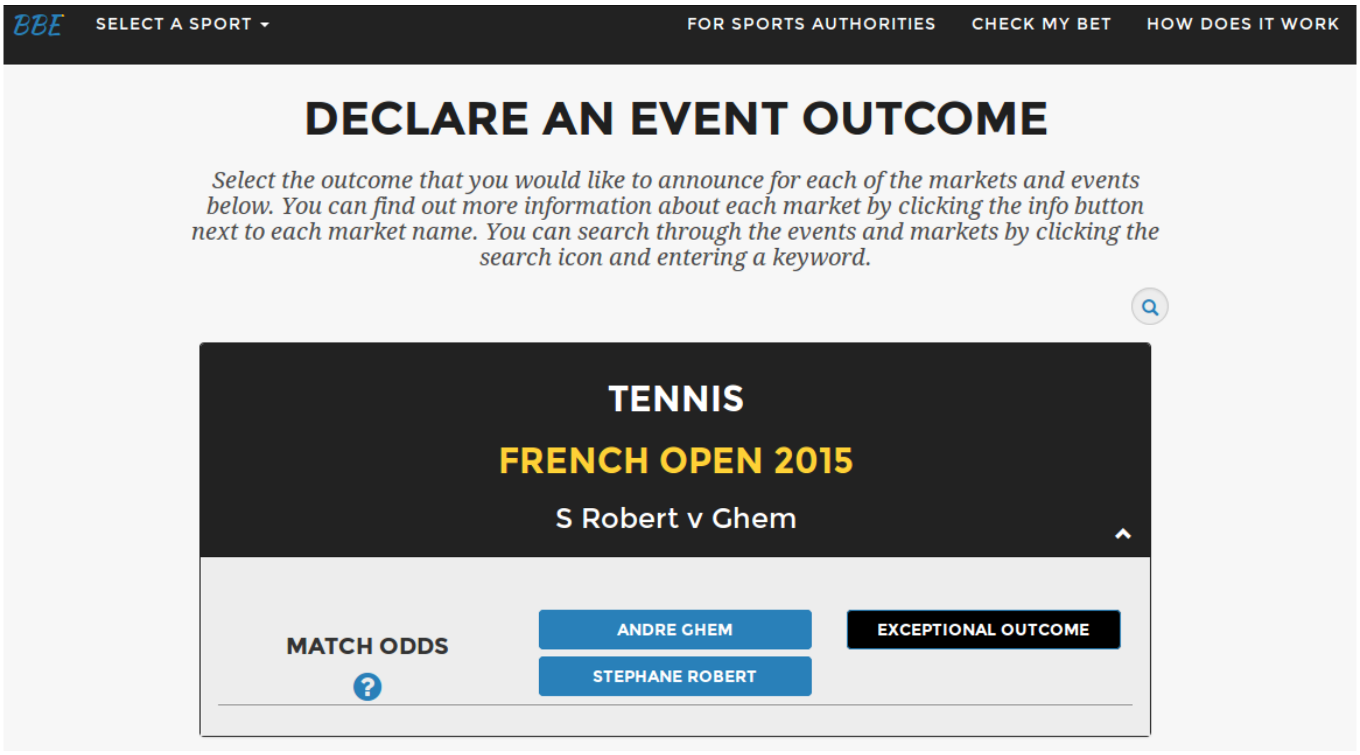
You need the passphrase to access the bet hash.

*****

Figure 57: A user entering the pass-phrase for their bet proof.

5.3 Proof of Outcome

We assign the responsibility for announcing event outcomes to the sports authorities for that event. When an event finishes, the authority can come to the *Bitcoin Betting Exchange* and select the *"For Sports Authorities"* section on the navigation bar. This will bring up a web page that allows the various sports authorities to see the currently unsettled markets and view the possible ways in which to settle those markets. Figure 58 shows a screen-shot of this page. As can be seen in the screen-shot, there is currently only one market that is yet to be settled. This is a *Match Odds* market for a tennis event, *S Robert v Ghem* in the *French Open 2015*.



DECLARE AN EVENT OUTCOME

Select the outcome that you would like to announce for each of the markets and events below. You can find out more information about each market by clicking the info button next to each market name. You can search through the events and markets by clicking the search icon and entering a keyword.

TENNIS

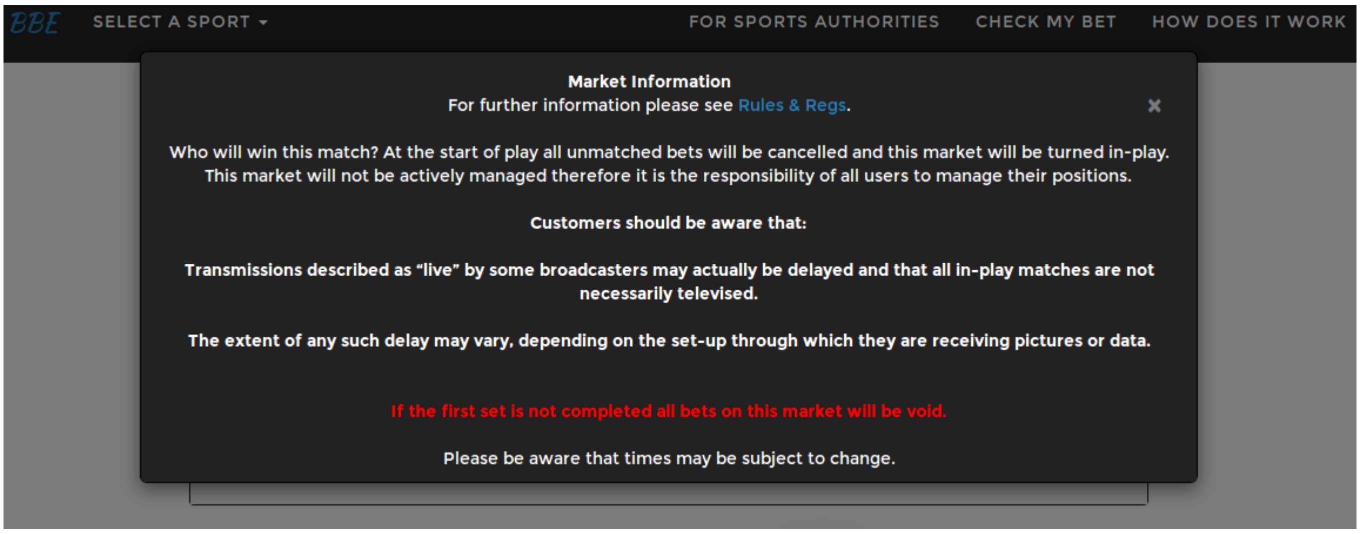
FRENCH OPEN 2015

S Robert v Ghem

Figure 58: The page for sports authorities that shows the currently unsettled markets.

When a sports authority comes to this page they can see the various outcomes that can be announced for each market. In order to announce the correct result for each unsettled market they can click on the *"?"* button to bring up more information about the market. Figure 59 shows a screen-shot of the additional information displayed for the *Match Odds* market. Having read the description and rules surrounding the market, the sports authority then has enough information to be able to decide the appropriate settlement for that market. In the case of a normal market settlement, the sports authority can select to declare one of the outcomes. This will open up a pop-up box displaying the *OP_RETURN* that the sports authority should announce. Figure 60 shows a screen-shot of the pop-up box displayed when *Andre Ghem* is selected the winner. This pop-up box also provides a quick link for the sports authority to verify that the hash displayed is correct for the selected outcome. This takes them to *online-convert.com* where they can hash the outcome data manually to be sure.

In addition to settling a market normally, sports authorities also have the option to settle a market with an *"Exceptional Outcome"*. This means that having read the rules of the market, the sports authority feels that none of the normal outcomes would be fair to the parties involved. Instead, they can choose to declare an *"Exceptional Outcome"*. In this case the payments of the bettors will be returned to them and the market will be declared void. This is required when unforeseen and exceptional circumstances occur, for example, if an event is terminated prematurely or cancelled. Figure 61 shows a screen-shot of the pop-up box displayed when an *Exceptional Outcome* is selected.



Market Information

For further information please see Rules & Regs.

Who will win this match? At the start of play all unmatched bets will be cancelled and this market will be turned in-play. This market will not be actively managed therefore it is the responsibility of all users to manage their positions.

Customers should be aware that:

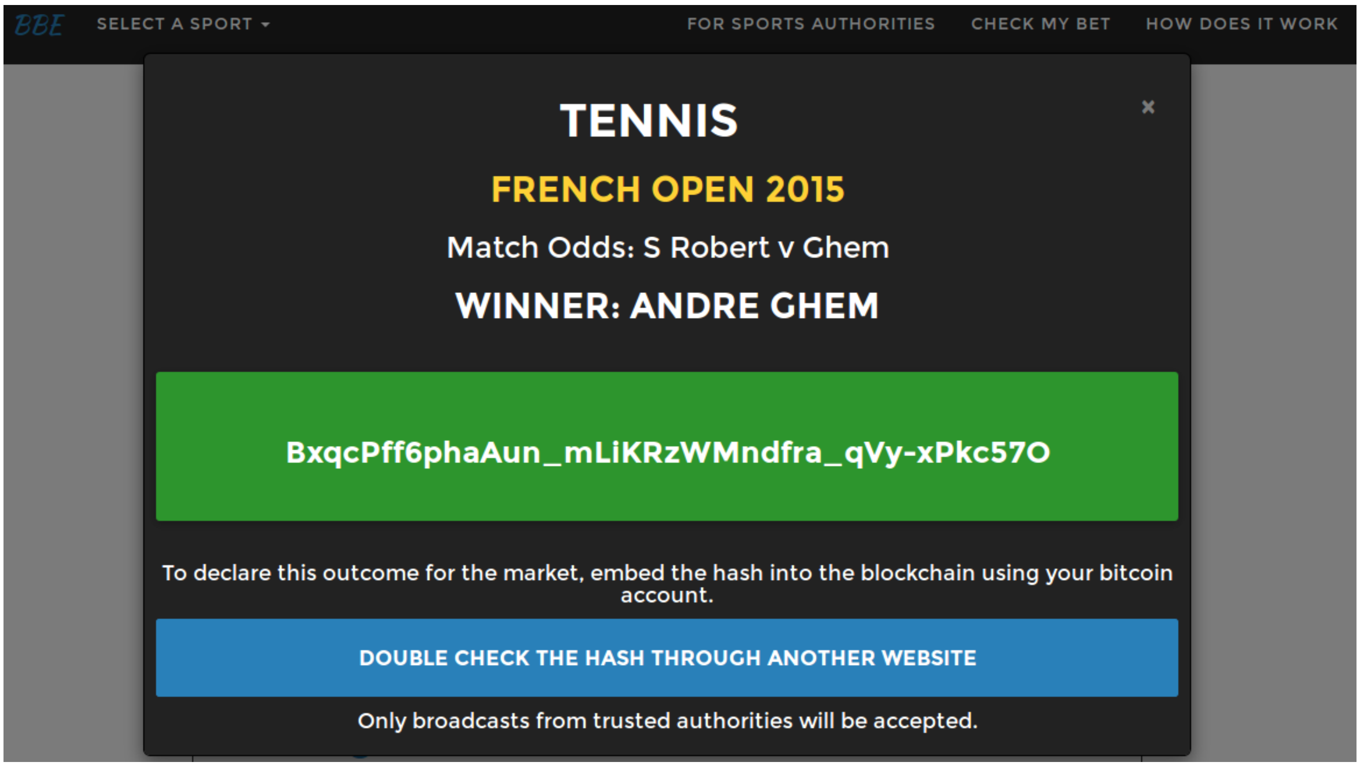
Transmissions described as “live” by some broadcasters may actually be delayed and that all in-play matches are not necessarily televised.

The extent of any such delay may vary, depending on the set-up through which they are receiving pictures or data.

If the first set is not completed all bets on this market will be void.

Please be aware that times may be subject to change.

Figure 59: Additional information regarding the match odds market.



TENNIS

FRENCH OPEN 2015

Match Odds: S Robert v Ghem

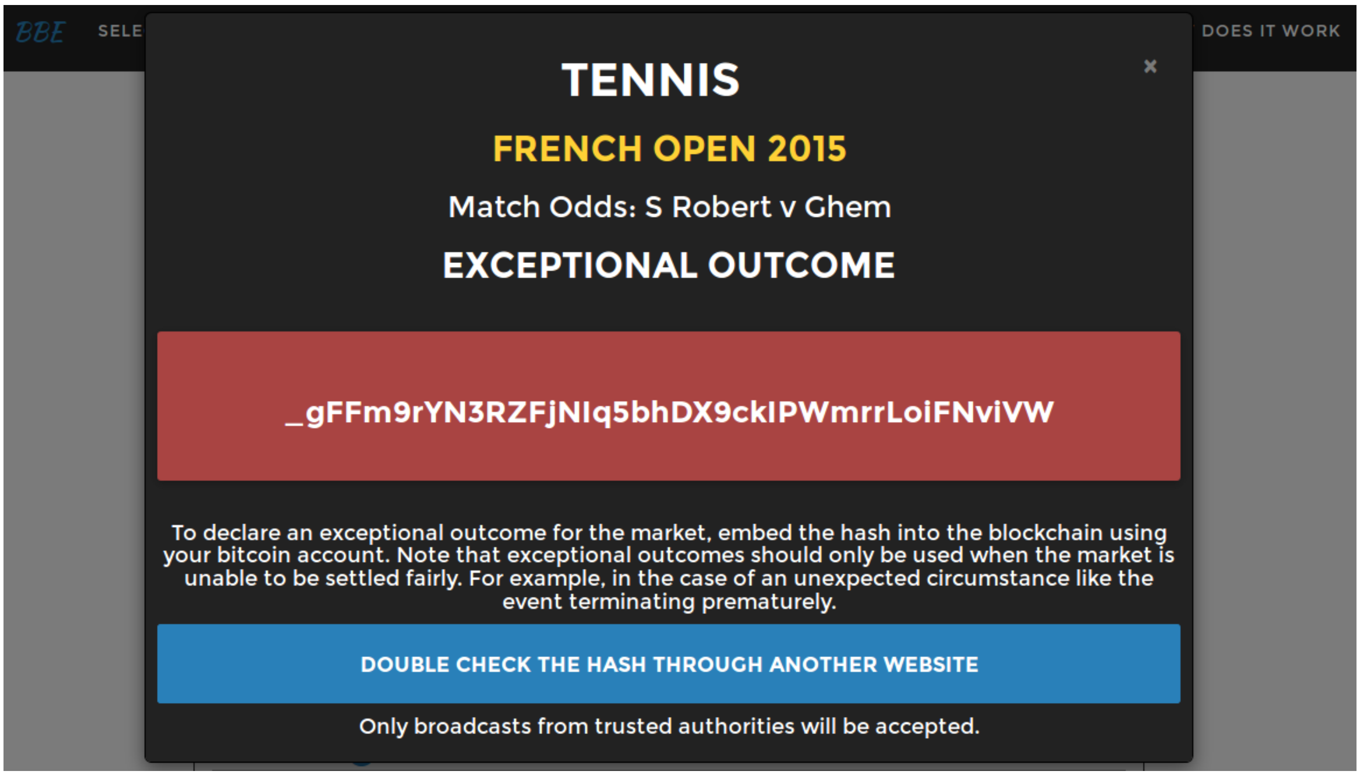
WINNER: ANDRE GHEM

BxqcPff6phaAun_mLiKRzWMndfra_qVy-xPkc57O

To declare this outcome for the market, embed the hash into the blockchain using your bitcoin account.

Only broadcasts from trusted authorities will be accepted.

Figure 60: The OP_RETURN for the sports authority to broadcast (declaring *Andre Ghem* the winner).



TENNIS

FRENCH OPEN 2015

Match Odds: S Robert v Ghem

EXCEPTIONAL OUTCOME

_gFFm9rYN3RZFjNIq5bhDX9ckIPWmrrLoiFNviVW

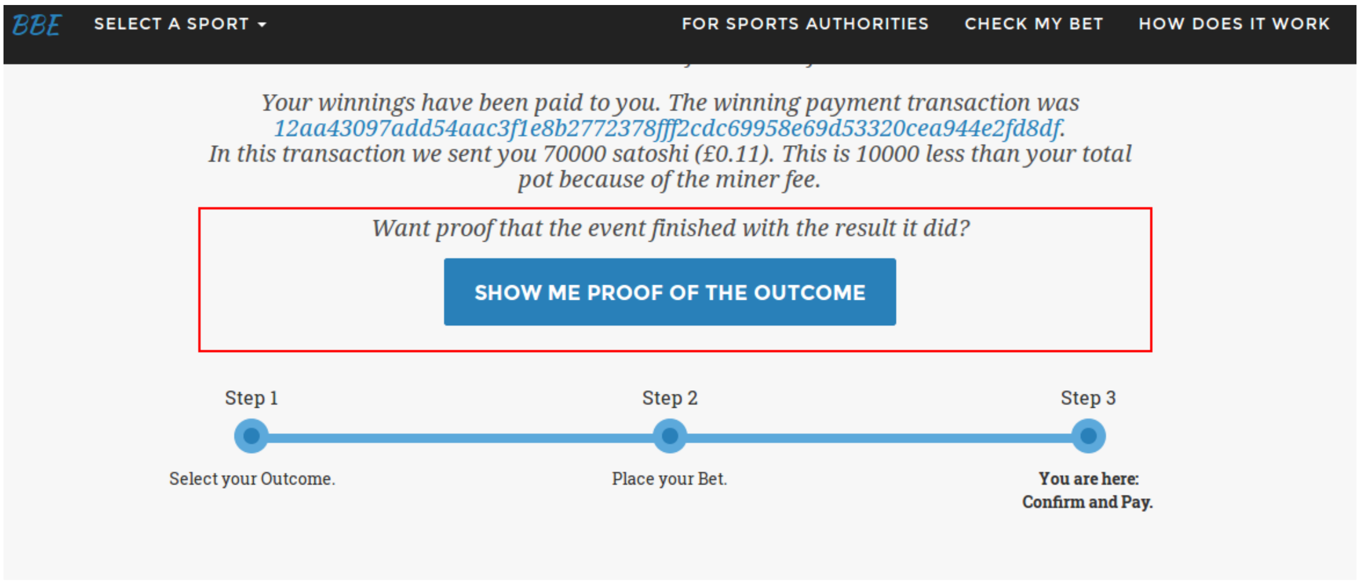
To declare an exceptional outcome for the market, embed the hash into the blockchain using your bitcoin account. Note that exceptional outcomes should only be used when the market is unable to be settled fairly. For example, in the case of an unexpected circumstance like the event terminating prematurely.

DOUBLE CHECK THE HASH THROUGH ANOTHER WEBSITE

Only broadcasts from trusted authorities will be accepted.

Figure 61: The OP_RETURN for the sports authority to broadcast (declaring an *Exceptional Outcome*).

Note that in order for these markets to be settled the sports authority needs to embed the selected hash into the *Blockchain* using an *OP_RETURN* transaction. As discussed in 4.1.6 this is very similar to how bet proofs are embedded into the *Blockchain*, except the private key announcing these *OP_RETURN* transactions belong to the sports authorities and not us. Because we only trust transactions that are signed by the public keys of the trusted sports authorities, it means that even if somebody else were to embed these hashes in the *Blockchain*, they would not be able to settle these markets. This means that we do not need to worry about displaying these hashes publicly.
When a sports authority declares one of these outcomes in the *Blockchain*, we settle the various bets that have been placed on the market and provide each bet with a link to validate the outcome of their bet. In the exact same manner that users are able to validate their bet address in the previous section, they can also validate the outcome of their bet. Figure 62 shows a screen-shot of the link presented to a user when their bet has been settled. The link is highlighted in red.

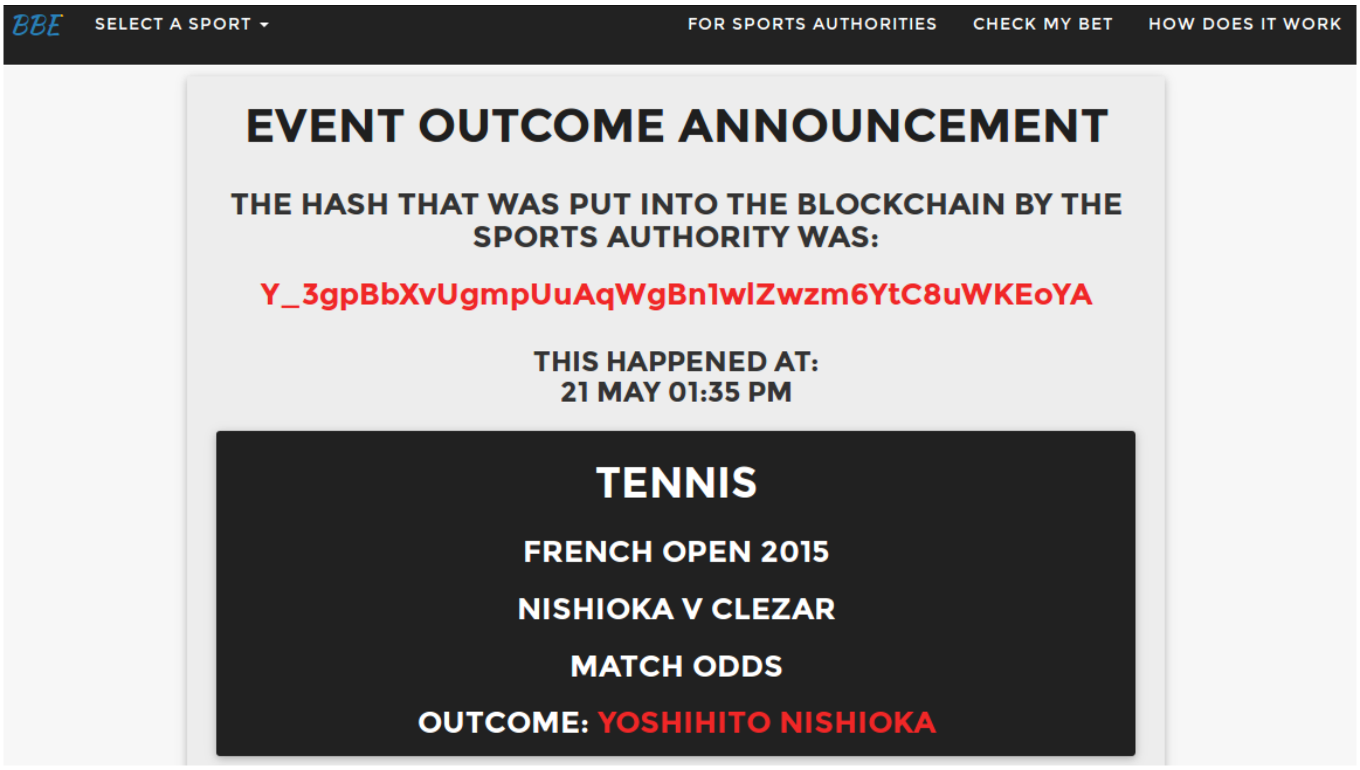


Your winnings have been paid to you. The winning payment transaction was 12aa43097add54aac3f1e8b2772378fff2cdc69958e69d53320cea944e2fd8df. In this transaction we sent you 70000 satoshi (£0.11). This is 10000 less than your total pot because of the miner fee.

Want proof that the event finished with the result it did?

Figure 62: The link on a bet page that takes users to the proof of outcome.

Clicking on the proof of outcome link on the bet page takes the user to the *Outcome Announcement* page. This page is very similar to the *Bet Announcement* page presented earlier, where users are able to verify that their bet address represents their exact bet. Similarly to the *Bet Announcement* page, the *Outcome Announcement* page explains *how the proof works*, provides links to *view the transaction* made by the sports authority, allows the user to *verify the hash* of the outcome (both in their browser and on an external site) and allows them to *download the proof* in pdf format for their own records. Figure 63 shows a screen-shot of the *Outcome Announcement* page.



EVENT OUTCOME ANNOUNCEMENT

THE HASH THAT WAS PUT INTO THE BLOCKCHAIN BY THE SPORTS AUTHORITY WAS:

Y_3gpBbXvUgmpUuAqWgBn1wlZwzm6YtC8uWKEoYA

THIS HAPPENED AT:
21 MAY 01:35 PM

TENNIS

FRENCH OPEN 2015

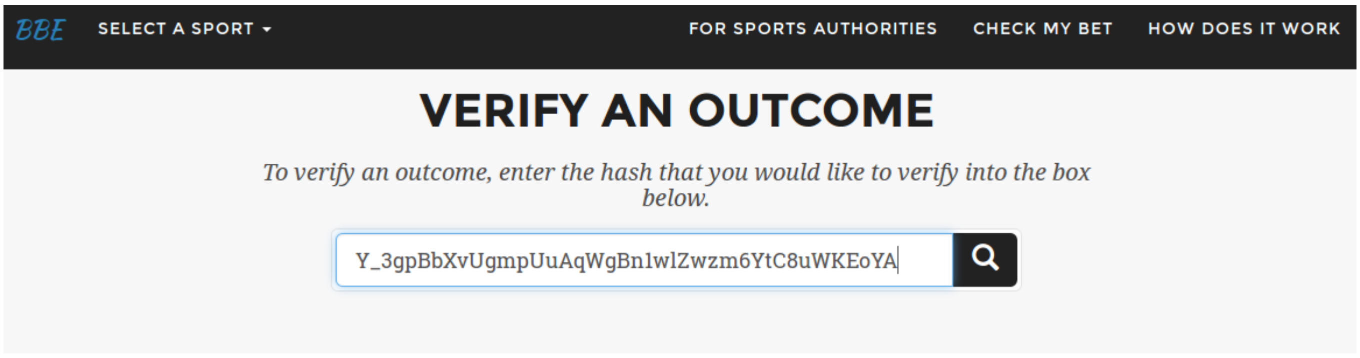
NISHIOKA V CLEZAR

MATCH ODDS

OUTCOME: YOSHIHITO NISHIOKA

Figure 63: The Outcome Announcement Page.

In addition to users being able to verify the outcome of their bet by going through their bet page, they also have the option to look up an outcome proof using the hash that was embedded into the *Blockchain* by the sports authority. This provides quick access to the proof without needing to go via the bet page. Figure 64 shows a user searching for an outcome proof using the *OP_RETURN* that was announced. Note that when users wanted to look up their bet proof via this page, they were required to enter the pass-phrase for their bet. In this case a pass-phrase is not required because revealing what an outcome announcement means does not reveal anything individual users, only how a market was settled.



VERIFY AN OUTCOME

To verify an outcome, enter the hash that you would like to verify into the box below.

Y_3gpBbXvUgmpUuAqWgBn1wlZwzm6YtC8uWKEoYA

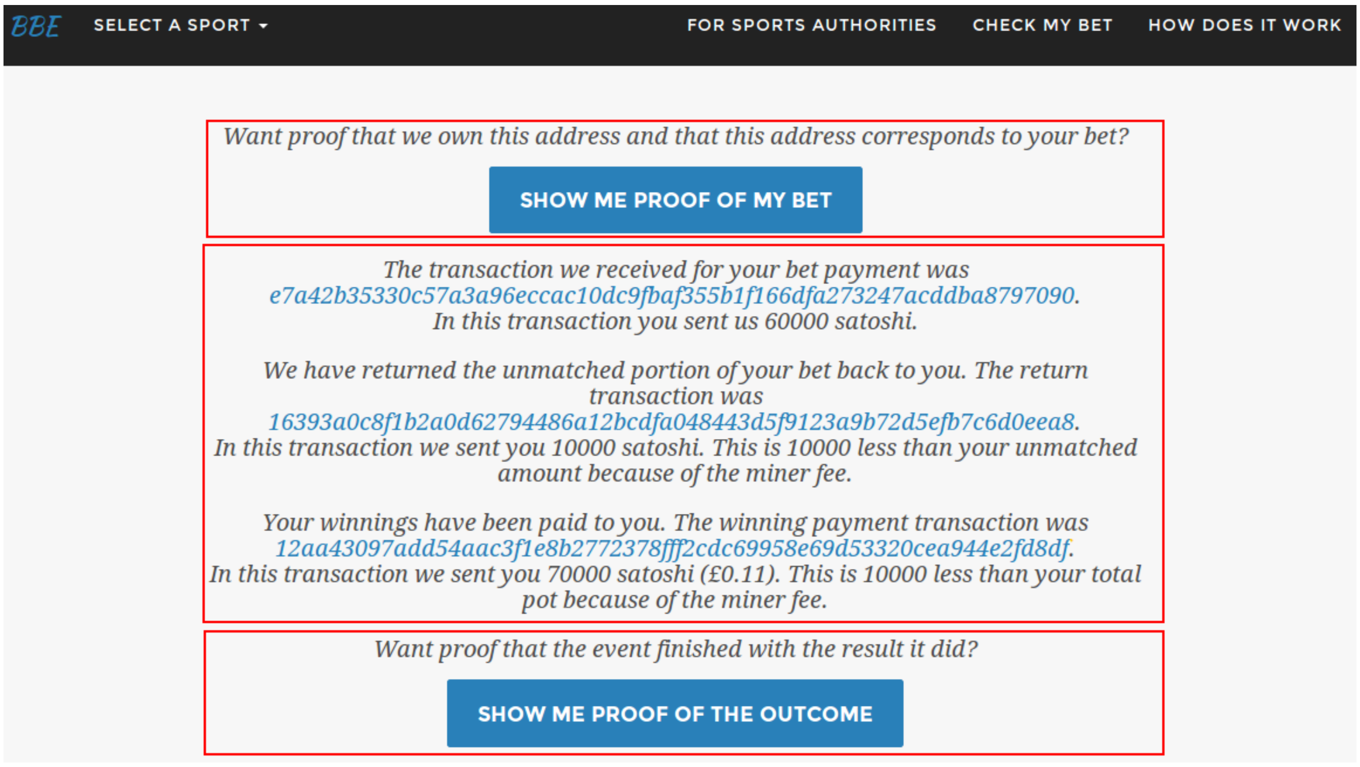
Figure 64: A user searching for an outcome proof via the *"Check My Bet"* page.

5.4 Audit Trail

The *Bitcoin Betting Exchange* makes it easy for users to track their bets and see the individual steps taken by both parties at various stages throughout the bet. It does this by displaying an *audit trail* at the bottom of the bet page. As actions occur, this *audit trail* is updated with all the pieces of information the user needs to effectively verify the current status of their bet. For example, if we take a look at the figure 65 we can see a complete *audit trail*, or list of steps taken, for a bet that was settled and won. The bet in this example is the winning bet we previously looked at, a summary of which can be seen in figure 66.

As previously discussed, in this example, 60,000 satoshi was paid by the user, and only 40,000 was matched. This means that 20,000 should have been returned to the user at the start of the event. Likewise, because the bet was made at odds of 2.0, the winner should have received 80,000 satoshi (or 70,000 when you take into account the miner fee). The *audit trail* shows us all this information in a step by step process. The very first step shown is that of the original bet proof published by the exchange. Once the user has verified the proof and is happy with it, they can then send their payment to the bet address. When a payment has been received by the exchange, the payment transaction is shown as the next step. Clicking the link provided takes the user directly to the transaction in *Blocktrail.com*. Figure 67 shows a screen-shot of the original bet payment. Highlighted in red we can see that the payment sent 60,000 satoshi to the bet address.

In this example, only 40,000 satoshi was matched, meaning when the event began 20,000 satoshi should have been returned to the user. When a transaction returning an unmatched bet portion is created by the exchange, this is displayed in the *audit trail*, with a link to the transaction. This is so that the user can verify the payment. We explain, as we did previously, that 10,000 satoshi is required for the miner fee when a return transaction is generated, meaning only 10,000 satoshi was sent back to the user. Figure 68 shows a screen-shot of the unmatched amount returned. Highlighted in red we can see that the transaction sent 10,000 satoshi back to the users address and left 40,000 satoshi in the bet address as the matched portion.



Want proof that we own this address and that this address corresponds to your bet?

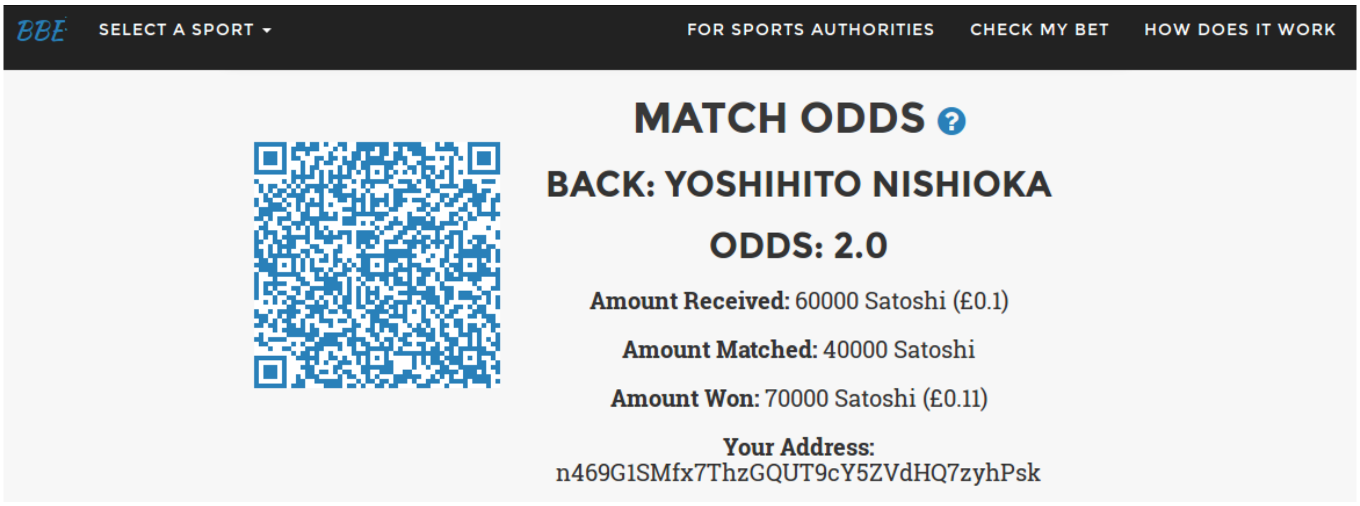
The transaction we received for your bet payment was e7a42b35330c57a3a96eccac10dc9fbaf355b1f166dfa273247acddba8797090. In this transaction you sent us 60000 satoshi.

We have returned the unmatched portion of your bet back to you. The return transaction was 16393a0c8f1b2a0d62794486a12bcdfa048443d5f9123a9b72d5efb7c6d0eea8. In this transaction we sent you 10000 satoshi. This is 10000 less than your unmatched amount because of the miner fee.

Your winnings have been paid to you. The winning payment transaction was 12aa43097add54aac3f1e8b2772378fff2cdc69958e69d53320cea944e2fd8df. In this transaction we sent you 70000 satoshi (£0.11). This is 10000 less than your total pot because of the miner fee.

Want proof that the event finished with the result it did?

Figure 65: The audit trail for a winning bet.



MATCH ODDS

BACK: YOSHIHITO NISHIOKA

ODDS: 2.0

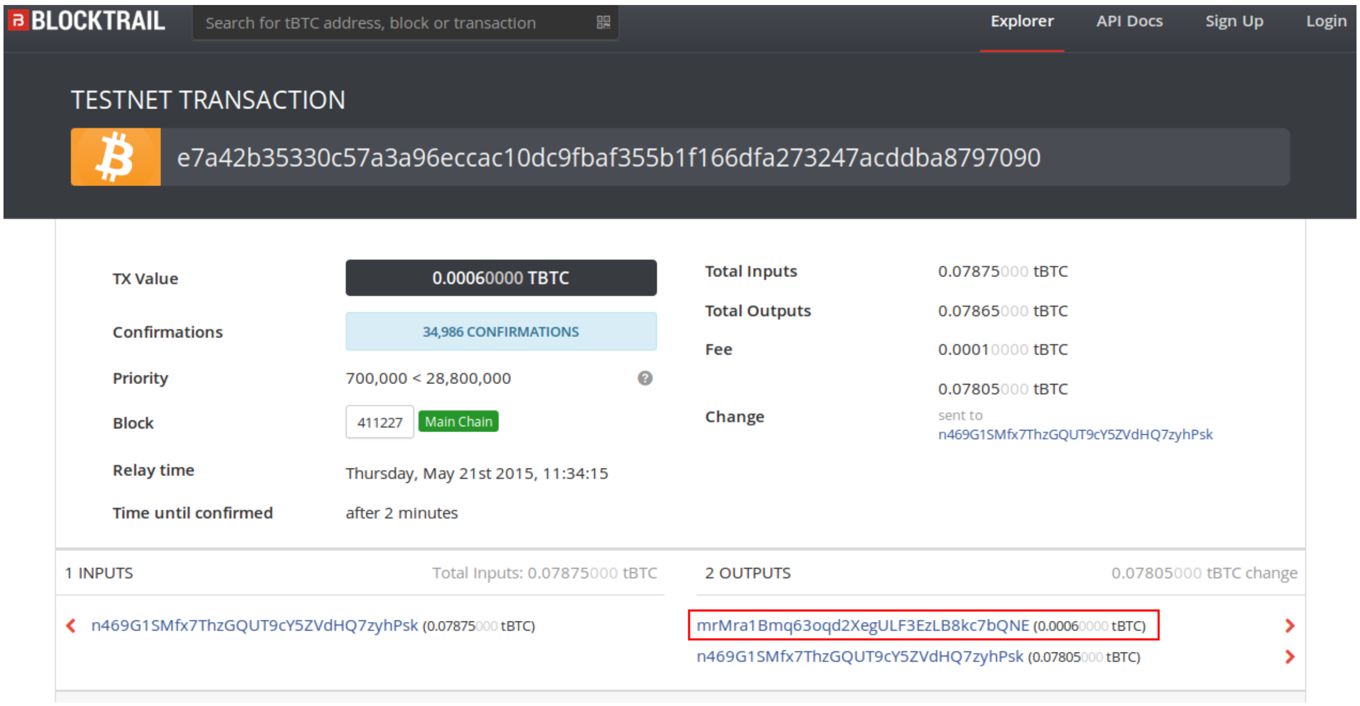
Amount Received: 60000 Satoshi (£0.1)

Amount Matched: 40000 Satoshi

Amount Won: 70000 Satoshi (£0.11)

Your Address: n469G1SMfx7ThzGQUT9cY5ZVdHQ7zyhPsk

Figure 66: The bet page for a winning bet.



TESTNET TRANSACTION

e7a42b35330c57a3a96eccac10dc9fbaf355b1f166dfa273247acddba8797090

TX Value	0.00060000 TBTC	Total Inputs	0.07875000 tBTC
Confirmations	34,986 CONFIRMATIONS	Total Outputs	0.07865000 tBTC
Priority	700,000 < 28,800,000	Fee	0.00010000 tBTC
Block	411227 Main Chain	Change	0.07805000 tBTC sent to n469G1SMfx7ThzGQUT9cY5ZVdHQ7zyhPsk
Relay time	Thursday, May 21st 2015, 11:34:15		
Time until confirmed	after 2 minutes		

1 INPUTS — Total Inputs: 0.07875000 tBTC

n469G1SMfx7ThzGQUT9cY5ZVdHQ7zyhPsk (0.07875000 tBTC)

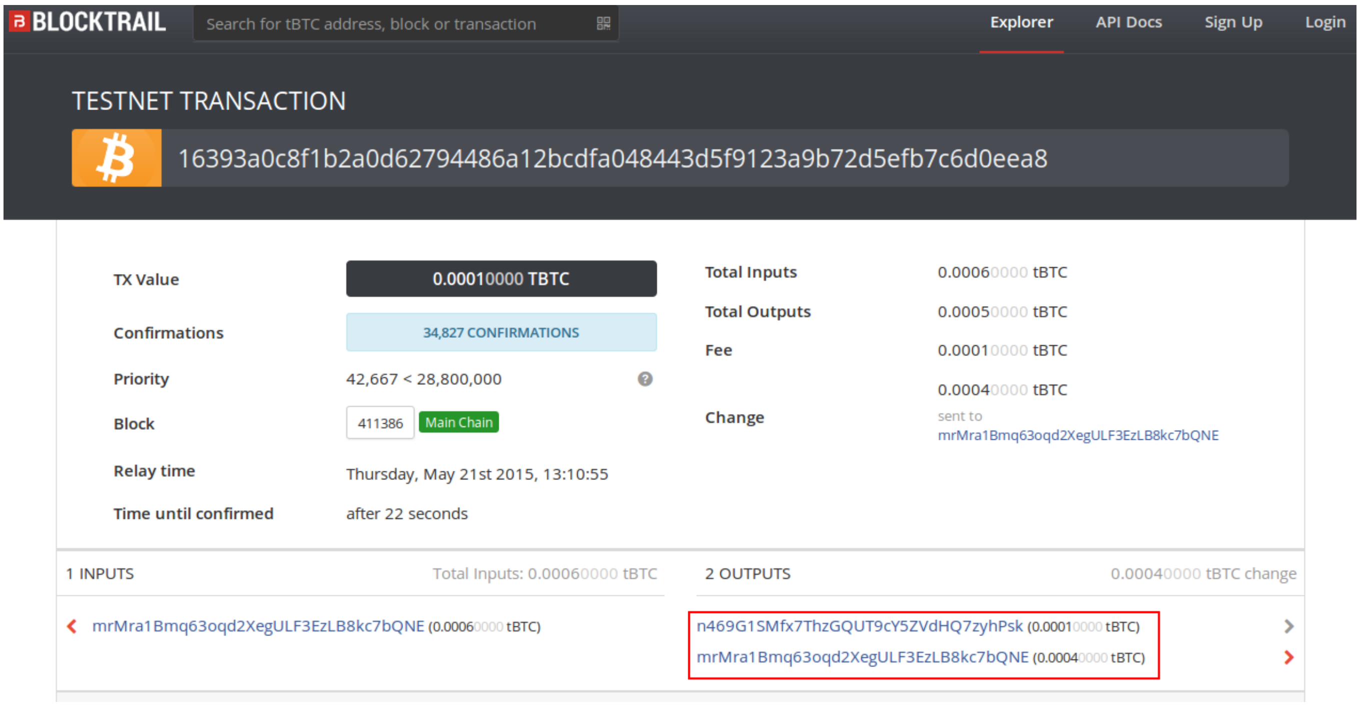
2 OUTPUTS — 0.07805000 tBTC change

mrMra1Bmq63oqd2XegULF3EzLB8kc7bQNE (0.00060000 tBTC)

n469G1SMfx7ThzGQUT9cY5ZVdHQ7zyhPsk (0.07805000 tBTC)

Figure 67: The original payment, of 60,000 satoshi, received for the bet.

When the event is finished and the outcome is announced, the user is then paid their winnings, an amount of 70,000 satoshi. Included in the *audit trail* is a link to the transaction that paid the user's winnings. Figure 69 shows a screen-shot of this payment. Highlighted in red we can see that the payment sent 70,000 satoshi (in total, made up of 59,999 + 10,0001) to the user's address. Finally, the last piece of information displayed in the *audit trail* is that of the outcome proof, so that if the user is not happy with how the market was settled (e.g. they lost), they can see the proof of outcome and verify it for themselves.


BLOCKTRAIL Search for tBTC address, block or transaction Explorer API Docs Sign Up Login

TESTNET TRANSACTION

16393a0c8f1b2a0d62794486a12bcdfa048443d5f9123a9b72d5efb7c6d0eea8

TX Value	0.00010000 TBTC	Total Inputs	0.00060000 tBTC
Confirmations	34,827 CONFIRMATIONS	Total Outputs	0.00050000 tBTC
Priority	42,667 < 28,800,000	Fee	0.00010000 tBTC
Block	411386 Main Chain	Change	0.00040000 tBTC sent to mrMra1Bmq63oqd2XegULF3EzLB8kc7bQNE
Relay time	Thursday, May 21st 2015, 13:10:55		
Time until confirmed	after 22 seconds		

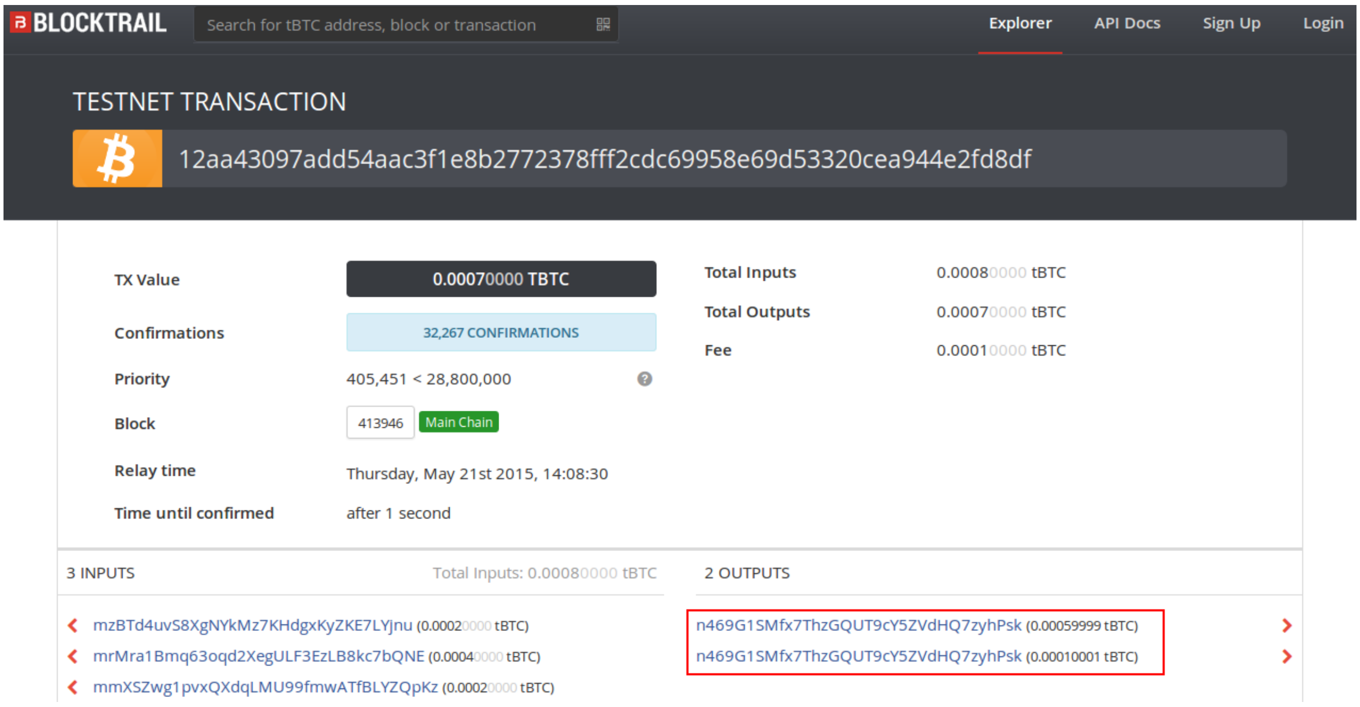
1 INPUTS Total Inputs: 0.00060000 tBTC

mrMra1Bmq63oqd2XegULF3EzLB8kc7bQNE (0.00060000 tBTC)

2 OUTPUTS 0.00040000 tBTC change

n469G1SMfx7ThzGQUT9cY5ZVdHQ7zyhPsk (0.00010000 tBTC)
mrMra1Bmq63oqd2XegULF3EzLB8kc7bQNE (0.00040000 tBTC)

Figure 68: The unmatched amount returned to the user, 10,000 satoshi and 40,000 satoshi left in the bet address.


BLOCKTRAIL Search for tBTC address, block or transaction Explorer API Docs Sign Up Login

TESTNET TRANSACTION

12aa43097add54aac3f1e8b2772378fff2cdc69958e69d53320cea944e2fd8df

TX Value	0.00070000 TBTC	Total Inputs	0.00080000 tBTC
Confirmations	32,267 CONFIRMATIONS	Total Outputs	0.00070000 tBTC
Priority	405,451 < 28,800,000	Fee	0.00010000 tBTC
Block	413946 Main Chain		
Relay time	Thursday, May 21st 2015, 14:08:30		
Time until confirmed	after 1 second		

3 INPUTS Total Inputs: 0.00080000 tBTC

mzBTd4uvS8XgNYkMz7KHdgxKyZKE7LYjnu (0.00020000 tBTC)
mrMra1Bmq63oqd2XegULF3EzLB8kc7bQNE (0.00040000 tBTC)
mmXSZwg1pvxQXdqLMU99fmwATfBLYZQpKz (0.00020000 tBTC)

2 OUTPUTS

n469G1SMfx7ThzGQUT9cY5ZVdHQ7zyhPsk (0.00059999 tBTC)
n469G1SMfx7ThzGQUT9cY5ZVdHQ7zyhPsk (0.00010001 tBTC)

Figure 69: The payment sent to the user, 70,000 satoshi in total, for winning their bet.

6. Implementation

In this chapter we discuss the general implementation details of the *Bitcoin Betting Exchange*. We explore the overall architecture, detailing and discussing the user-facing web server and our smart, private key store. We explain how our application interacts with the outside world, such as how it communicates with the *Bitcoin* network and how it provides real-time event data. We also highlight several interesting challenges that arose when implementing our solution, and discuss the approaches we took to solve those problems.

6.1 Architecture

The *Bitcoin Betting Exchange* is comprised of two distinct parts. Figure 70 highlights these parts and outlines the general architecture of our solution. The first part of the *Bitcoin Betting Exchange* is a public facing web application that users visit to bet on sports events and markets. In the diagram we can see that when Alice and Bob visit the betting exchange online, they do so by connecting directly to the server that runs the web application. This is the web application that users primarily know as the *Bitcoin Betting Exchange*. The second distinct part of our solution is a software implementation that mimics the hardware based private key store we outlined in section 4.2. Due to the specific nature of this project it was considered out of scope to build a hardware prototype of the private key store. As such, we have used a secondary, stripped-down, web application to mimic the behaviour of the key store. When the betting exchange needs to communicate with the private key store, it does so by sending requests directly to the server that is running the second web application.

Both web applications have their own databases that they communicate with in order to store information specific to them. The web application that mimics the private key store has been designed in such a way as to ignore all requests from the outside world, except those made by the betting exchange. The private key store therefore only communicates with the betting exchange and its own database. All other connections are refused. This is so that the entire system is implemented in a way to reflect the original hardware proposal.

In addition to its own database, the web application for the betting exchange also communicates with several external *APIs* (application programming interfaces). These *APIs* allow the betting exchange to retrieve the latest sport and event data, communicate with the *Bitcoin* network, and lookup the latest *Bitcoin* price index and conversion rates.

One thing to note about our architecture is that all communication paths operate under encryption, using *SSL* (secure sockets layer). For example, when Alice and Bob connect to the betting exchange in their browser, the betting exchange operates over *HTTPS* (*HTTP* over *SSL*). Likewise, when the betting exchange communicates with the private key store, this too is over *HTTPS*. All communication with external *APIs* are over *HTTPS* and the connections between the web applications and their databases use *SSL*. Using an encrypted connection is essential when running a service that operates over the Internet. This is because encrypted communication paths prevent an attacker from simply reading the information that flows between the two end points.

For example, if the communication path between Alice and the betting exchange is not encrypted, an attacker might be able to monitor all the *url* requests she makes. This would allow the attacker to read all the information embedded in the *url*, and, for example, they would be able to find the unique bet address for Alice's bet (as this is embedded directly into the *url* when a bet is requested). By encrypting the *url* an attacker can only see that Alice is making a request to the *Bitcoin Betting Exchange*, but they cannot see what pages or paths she is visiting or what operations she is performing. This is extremely important for anonymity.

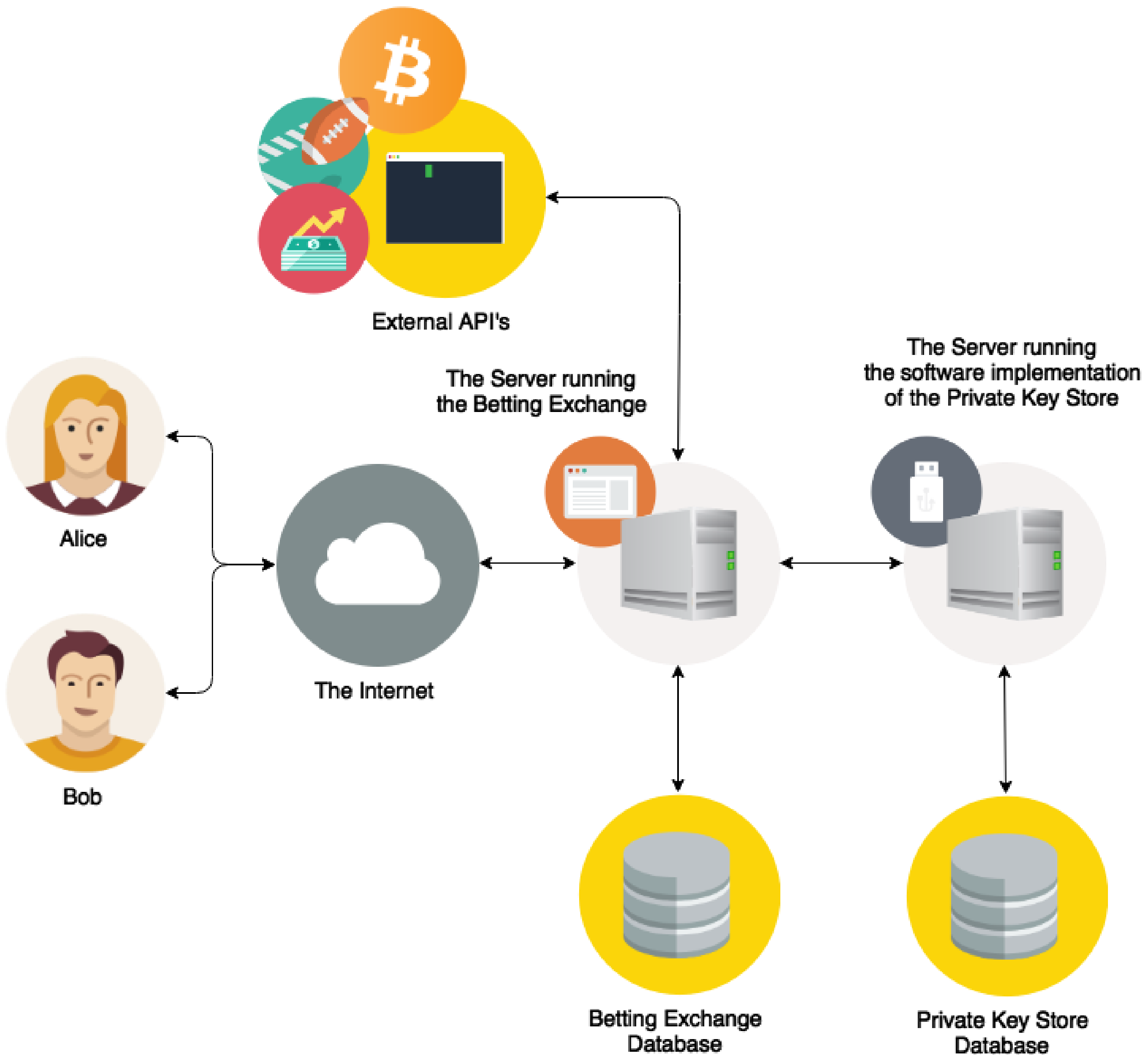


Figure 70: The architecture of the *Bitcoin Betting Exchange.*

6.2 The Bitcoin Betting Exchange

6.2.1 Infastructure & Platform

When deciding on what tools and platforms to use to build our user-facing web application, there were several factors that heavily influenced our decisions. Because the web application would be the first prototype of our proposed solution, we required something that would allow us to iterate rapidly and react to any challenges that might arise. As such, it should be something that is well suited to an *agile* work flow, being well supported and documented, with a large community. The platform we choose should also allow us to produce a solution that is user-friendly, attractive and stable. In addition it should already be in wide-scale use, provide a strong focus on testing and allow us to easily host it online, at a low cost.

After looking at various options we ultimately decided that *Ruby on Rails*[106] would be the most appropriate choice for us. This is because *Ruby on Rails* is a free, open-source platform that has been around for many years, with a large community and a wide-array of publicly provided libraries, called *Ruby Gems*. *Ruby on Rails* is well suited to an *agile test-driven development* work-flow by providing templates of many kinds to allow rapid iteration, as well supporting many test suites. *Ruby on Rails* is powered by *Ruby*, a *dynamic* and *general purpose* programming language that supports many programming paradigms and is easy to read and understand. In addition, *Ruby On Rails* already provides support for many different types of databases, such as *Postgres*, *MongoDB* and *Oracle*. This makes it ideal for our requirements.

We therefore implemented our web-application in *Ruby on Rails*. One of the great things about using *Ruby on Rails* is that *Rails* applications are typically very easy to host online, as there any many different services who provide support specifically for *Rails* applications. Furthermore, the process of deploying a *Rails* application is typically fairly straightforward and simple. We therefore decided to host our application online using *Heroku*, *"A platform as a service (PaaS) that enables developers to build and run applications entirely in the cloud"* [66]. *Heroku* is a cloud platform that provides online web hosting for many different types of web applications. Internally, *Heroku* runs on *Amazon EC2*, the *Amazon Elastic Compute Cloud* [65] platform and so, just like *EC2*, it support scaling of hosting and different price tiers for users.

The very first price tier that *Heroku* offers is completely free of cost. Although this is ideal for a prototype application, it does come with restrictions on the amount of requests it can serve per month and the size of the database it supports. Because the focus of our application is to produce a working prototype and not a solution that is commercially optimized, this tier was adequate for our needs.

When considering the type of database we would need for the betting exchange, there was a distinct trade-off to be made between *relational* databases and *non-relational* databases. Given that our betting exchange would only need to store a limited amount of sport and bet data it was clear that choosing to use a *non-relational* database, such as *nosql*, would be unnecessary. The reasons behind this decision were that, firstly, the type of data we were going to store was naturally structured, fitting in well with the constrains of a *relational* database quite nicely (e.g. sports have many events, and events have many markets and so on). Secondly, because the amount of data we planned on storing was not excessive, with no storage of user data required, sacrificing structure for scalability seemed unnecessary. This was because given that our primary motivation behind the project was to build a prototype, and not a commercially scalable solution, the benefits a *non-relational* database would provide would be minimal. As such, we therefore opted to use a *relational* database to back our web application.

Heroku provide free support for *Postgres* [59] databases when using *Ruby on Rails* applications. *Postgres* is an open-source *relational* database system that has been around for many years. It is widely used, well supported and generally considered quite stable. Given that *Heroku* provide *Postgres* free of charge for their most basic tier, this was ideal for our requirements, and thus we opted to use a *Postgres* database for our web application.

In addition to hosting, *Heroku* also provide support for *continuous integration*. Using *Heroku*, you can setup a *continuous integration* and *deployment pipeline* that automatically tests and pushes new versions of the application directly to the users. This works simply by connecting the hosting server to a *GitHub* [53] repository. Given that version control is a fundamental requirement in any software development project, and that *Git* is very widely used and trusted, we used this feature to set up a *continuous integration* and *deployment pipeline* for our web application. This was especially useful given our adoption of an *agile* approach to development, as it allowed us to continuously roll-out of each iteration of the project. This provided us the ability to gain immediate feedback on the current state of the solution from users.

6.2.2 Live Event Data

In order to take bets on real sports events and markets, we needed to find a source of real-time sport data. Given that *Betfair* [13] is currently the world's largest online betting exchange and offers free API [97] access to their exchange for non-commercial use, it seemed appropriate to use their service. Not only has their API been around for a long time, but it is used by many different financial applications and systems worldwide, and so it is generally considered quite stable and well-documented. In addition to this, because *Betfair* is the most popular betting exchange online, it provides a very high degree of market liquidity. Integrating *Betfair* data into our solution would allow us in the future to provide bet matching across fiat currency. This would allow bets placed in *Bitcoin* to be matched with bets placed in fiat currency, and vice versa. This offers a higher degree of flexibility and increased market liquidity. We have already seen this done when we looked at *DirectBet*. *DirectBet* use the *Betfair* betting exchange to power their *Bitcoin* exchange, by matching bets across *Bitcoin* and normal fiat currencies.

As mentioned, one of the benefits of using *Ruby on Rails* is its large collection of public libraries, called *Ruby Gems*. Because *Betfair* is quite well known, there were already several *Ruby Gems* available online that provided *Betfair* API integration. This meant that we would not be required to manually construct API requests using JSON, but instead could use the methods and response objects provided by those gems. The *Ruby Gem* we used to integrate the *Betfair* API into our application was *betfair_api_ng_rails* [72], a gem built specifically for *Rails* applications. Using this gem and our *Betfair* API credentials, we were able to make API calls to *Betfair* to get various sports, events and market data.

Figure 71 below shows a code snippet of an API call that fetches the list of all available sports from *Betfair*. Note that the `BetfairApiNgRails` module in the snippet refers to the gem we are using to communicate with *Betfair*. In addition, a `MarketFilter` is used to filter the results of our API call. The `MarketFilter` we create in this example is empty, meaning that all available sports are returned. Further, we specify that the language, or locale, of the response should be in English. When we are given a response from *Betfair* we map across the list of response objects and extract the `event_type` object out of the response body.

```ruby
# Loads all of the event types (eg. sports) from Betfair
def get_event_types()
  event_type_results = BetfairApiNgRails.list_event_types(filter:
    BetfairApiNgRails::MarketFilter.new, locale: :en)

  event_types = event_type_results.map { |event_type_result| event_type_result.
  event_type }
  return event_types
end
```

Figure 71: A method that fetches all of the available sports from the *Betfair* API.

Due to the way that the *Betfair* API works and the limitations that exist on the frequency and size of client requests, we were required to build several database models and tables to store the sports data. The reason for storing the data locally was that, if the data was not stored in some form of local storage or database, the application would need to make API requests to *Betfair* on the fly. This would not only be slow for users visiting the site, given that multiple requests would be required for each page, but the solution would also not scale. This is because as you increase the number of users visiting the web application concurrently, you will very quickly reach the request limits of the API and therefore, requests will be refused and users will not be able to access the sport data. In addition, you will also be breaking the terms and conditions provided by the API regarding these limits. By storing the sport data locally, it would not only provide a more responsive web application, but make it much easier to associate bets with various events later on.

In order to store the various pieces of sport and market information, we were required to build several tables to model the data we received from *Betfair*. One of the benefits of using *Ruby on Rails* is its *Active Record Query Interface* [105], an interface that makes it easy to construct various database objects and create associations between those objects. By using *Active Record* we could communicate with the database using a much higher level of abstraction than that provided by SQL. This made writing the logic to manipulate the data much easier.

Figure 72 below shows a simplified entity relationship diagram of our database. As can be seen in the diagram, there are 5 unique database objects that make up a market and outcome. The first object is an *EventType*, which represents a unique sport, such as *Tennis* or *Cricket*. The second object is an *Event*, which represents a unique event for a single sport, such as a tennis match of *Roger Federer vs. Rafael Nadal*. An *EventType* has many *Events*, and each *Event* belongs to a single *EventType*. In addition, each *Event* may be associated with a single *Competition*, such as the event of *Roger Federer vs. Rafael Nadal* belonging to the *2015 Wimbledon* tournament. Likewise a *Competition* may contain several events. Furthermore, each *Event* has many *Markets*, with a single *Market* belonging to a single *Event*. A *Market* for a tennis event may be *"Winner: Who Will Win the Match?"* or *"First Set: Who Will Win the First Set?"*. Finally, each market has many *Runners*, or outcomes, and each *Runner* might belong to many *Markets*. For instance, *Roger Federer* is a runner in the market of *"Who Will Win the Match?"* for the *Roger Federer vs. Rafael Nadal* tennis event. Likewise, *Roger Federer* may also be a runner in a different market for a different match.

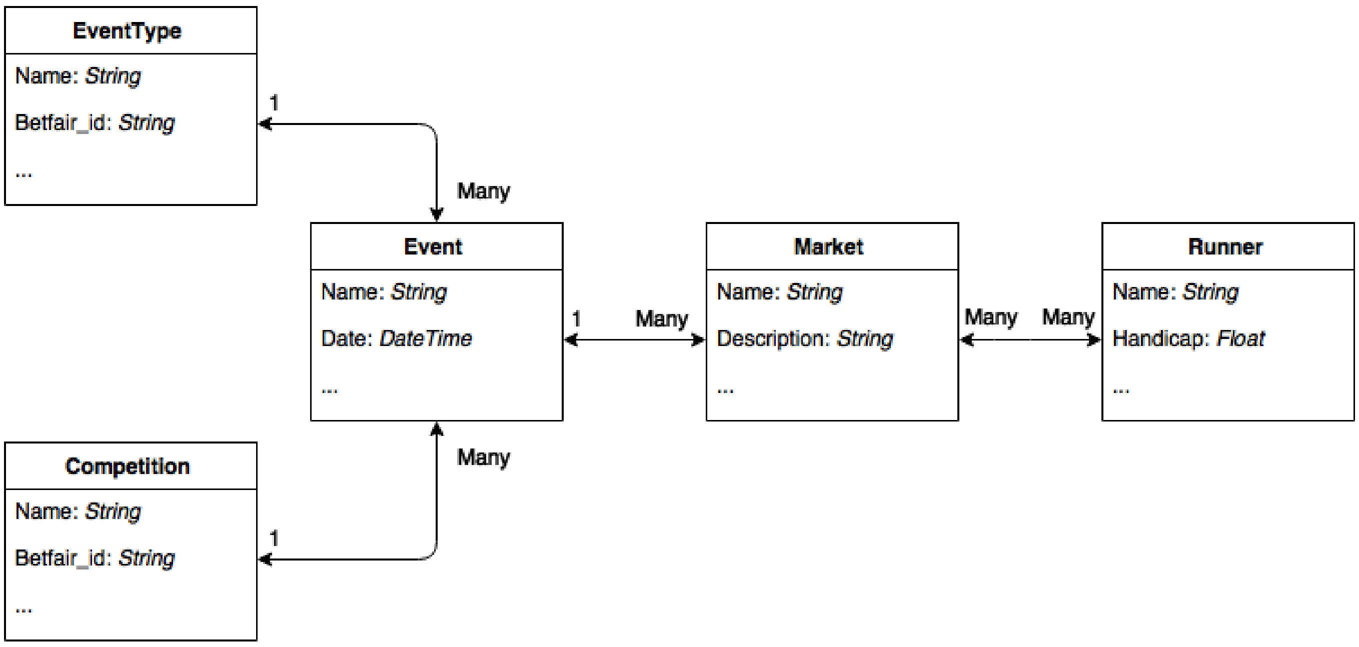


Figure 72: A simplified entity-relationship diagram of the database that stores the sport and market data.

Each entity in the diagram represents a single table in our database, with *one-to-many* associations implemented using foreign key constraints and *many-to-many* associations implemented using join tables. As can be seen in figure 72 each database object stores information specific to its function. For example, an *EventType* stores the name of the sport it represents, an *Event* stores the date and time at which it starts, and a *Runner* stores any handicap assigned to it. In addition, we store the unique `Betfair_id` of the object when we write it to our database. This is so that we are still able to map between objects in our own application, and objects in the *Betfair* store. Figure 72 shows a very simplified view of our database, with many fields removed for illustration purposes. For a complete entity relationship diagram of our database, refer to Appendix A.

At present our application supports around 20 different sports, and over 1000 different events at any given time. Figure 73 shows a screen-shot of the sports currently supported by the *Bitcoin Betting Exchange*.

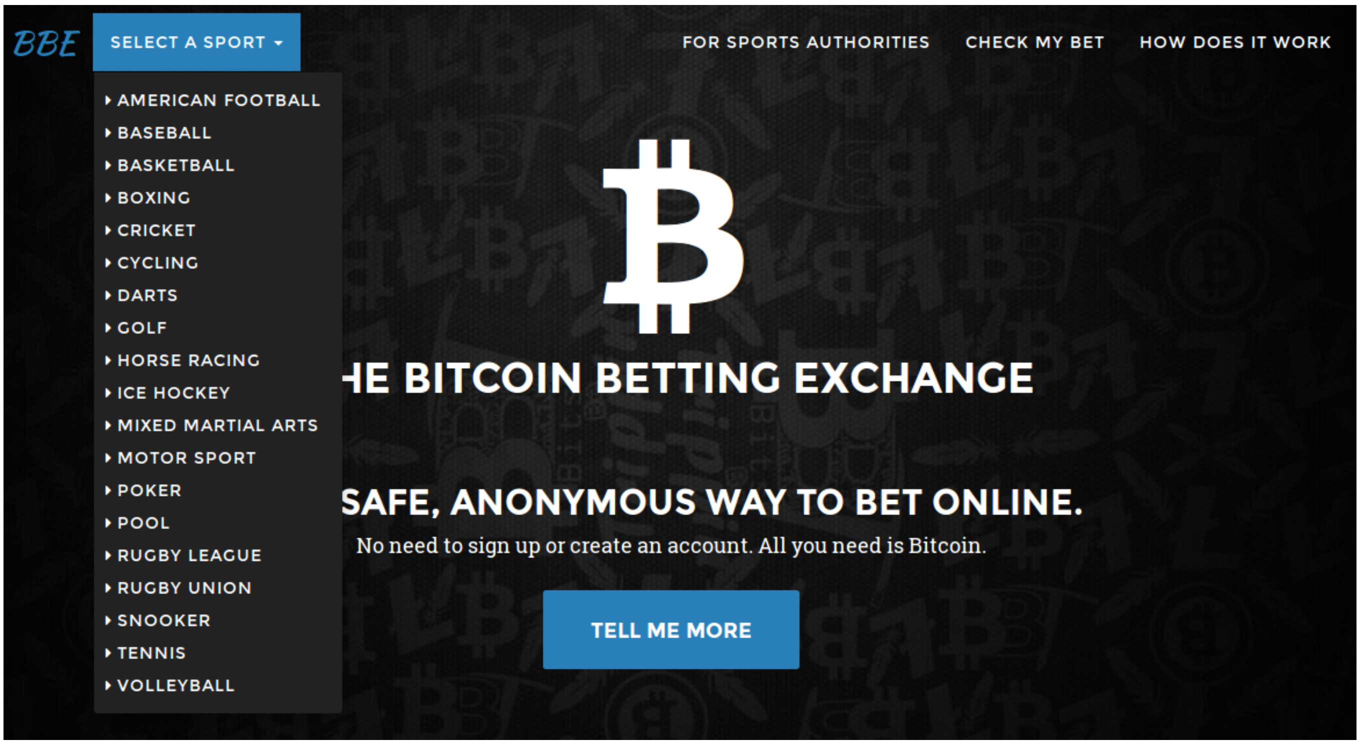


Figure 73: A screen-shot of all the available sports currently supported by the *Bitcoin Betting Exchange*.

In order to keep up to date with the current sports and market information available through *Betfair*, we need to continuously keep polling the *API* and re-syncing our database. Performing a complete re-sync of the database takes around an hour to complete. Given that the rate at which new events are announced is fairly low (e.g. every 1-2 days) we would only need to perform a complete re-sync of the database every day. When you consider that a re-sync puts additional load onto the database, it would make sense not to perform this operation too often, otherwise it might slow down the rate at which it could respond to user requests. As such, we needed to find a way to periodically schedule these updates at a time that the application is not under heavy load.

To do this, we employed the use of a well-known task scheduler built specifically for *Rails* applications, *rufus-scheduler* [77]. The *rufus-scheduler Ruby Gem* is a job scheduler for ruby that supports background scheduling of various tasks and jobs. It provides a very easy way to schedule repetitive tasks and supports *cron* format. As such, we decided to use *rufus-scheduler* to re-sync the database at around midnight every night in the UK. We chose midnight because it signifies the start of a new day and would be a time at which there is not much user activity occurring.

Figure 74 shows a code snippet of the scheduler that updates the database with new sport and market data every night. As can be seen in the figure the job runs every night at 5 minutes past midnight and calls the `fetch_and_update` method on the `EventAndMarketPopulator` module. We wrap this call in a database transaction using `ActiveRecord::Base.transaction` to prevent the database from being updated to an inconsistent state in the case that the job fails. We catch any failures, roll back the transaction and write the exception to the server log for later inspection.

Under failure, our application continues to work with the old data.

```ruby
# Update Betfair Data at 12:05 every night
scheduler.cron "5 0 * * *", :overlap => false, :mutex => "database" do
  begin
    ActiveRecord::Base.transaction do
      EventAndMarketPopulator.fetch_and_update
    end
  rescue => e
    Rails.logger.info "Error in EventAndMarketPopulator: #{e.inspect} " + Time.now.to_s
    Rails.logger.info "Back Trace: #{e.backtrace} "
  end
end
```

Figure 74: A scheduled job written using *rufus-scheduler* to update the database with new event and market data every night.

There were two interesting challenges we faced when integrating the *Betfair* API into our web application. The first was discovering a number of unfortunate bugs in the *betfair_api_ng_rails* gem. One of which was a missing `MarketFilter` attribute that allowed you to search for information relating to a specific market when an API call is made. In order to overcome this, we cloned the gem locally and added the `market_id` filter to the source code ourselves, before recompiling the gem. We then highlighted this issue to the original developers online. Figure 75 shows a code snippet of the class we modified. The attribute we added to the `MARKET_FILTER_ATTRS` field is at line 9. This allowed us to search by market ids when making an API call.

```ruby
 1  class MarketFilter < Api::Data::Base
 2    ...
 3    MARKET_FILTER_ATTRS = [
 4      :text_query,
 5      :exchange_ids,
 6      :event_type_ids,
 7      :event_ids,
 8      :competition_ids,
 9      :market_ids,  # Our solution - add the attribute to search by market ids
10      :venues,
11      ...
12      :market_start_time,
13      :with_orders
14    ]
15  end
```

Figure 75: Our fix to a missing attribute for a `MarketFilter` in *betfair_api_ng_rails* gem.

The second interesting challenge we faced was to do with *Heroku* deployment. As previously mentioned, online sports betting is illegal in the US, and by default *Heroku* deploy web applications using a web server based in the US. Unsuspectingly to us, when we first deployed the scheduler online, it failed every time the database task ran, displaying an error that *Betfair* could not be contacted. This was due to the fact that *Betfair* is not accessible in the US, as it solicits online sports betting, which is illegal. Therefore any and all requests to *Betfairs* APIs originating from an *ip address* in the US are blocked. In order to overcome this, we re-deployed the application to a server based locally in the UK. This also provided the added benefit that the web application responded much more quickly to local requests as there was less latency in the connection.

6.2.3 Integration with the Bitcoin Network

In order to allow our web application to communicate with the *Bitcoin* network (e.g. when it needs to broadcast a new transaction or check if a payment has been sent by the user) we make use of several different *Bitcoin APIs*. Instead of running our own *Bitcoin* node and having our application talk directly to that node, we communicate with the *Bitcoin* network through various online *APIs*. These *APIs* run their own *Bitcoin* nodes and communicate directly with the *Bitcoin* network. Choosing to use an existing *API* over running our own node reduces the amount of infrastructure we need to build, deploy and maintain. As such, it decreases the overall complexity of our design. In addition, because we generate and store our own private keys locally, and construct signed transactions ourselves, we only need to use these *APIs* to broadcast transactions to the rest of the world and to fetch existing transactions from the *Blockchain*. This means that we never expose any of our private keys or sensitive information to the *APIs*.

There are three separate *Bitcoin APIs* that we use to connect to the *Bitcoin* network. These are *Chain.com* [33], *Chain.so* [35] and *Test.Webbtc.com* [112]. The primary *API* we use is *Chain.com*. *Chain.com* is an *API* built specifically for web applications that use *Bitcoin*, providing both paid (commercial) and free access to the API. One of the benefits of using the *Chain.com* API is that it provides explicit support for fetching and decoding strings embedded in *OP_RETURN* transactions. Because our application uses *OP_RETURN* transactions to announce *bet proofs*, by using the *Chain.com* API we can easily query and look up the *OP_RETURN* transactions that we have announced into the *Blockchain*. This makes interfacing with the *Blockchain* much easier.

In addition, *Chain.com* also provides support for applications that use *Ruby*. They do this by providing a *Ruby Gem*, *chain-ruby* [34], that makes it simpler and more straight forward to communicate with their *API*. Figure 76 shows a code snippet of a method that fetches all of the *OP_RETURN* transactions created by a specific *Bitcoin* address, from the *Bitcoin* network. It does this by making an API call (`get_address_op_returns(...)`), to *Chain.com* with the given *Bitcoin* address. The `@bitcoin_chain_client` variable is an instance variable that represents the *Chain.com API* module provided by the *Ruby Gem*.

```ruby
# Gets all of the OP_RETURN transactions created by the given bitcoin address
def get_op_returns_for_address(bitcoin_address)
  return @bitcoin_chain_client.get_address_op_returns(bitcoin_address)
end
```

Figure 76: An API call to *Chain.com* that gets all of the *OP_RETURN* transactions created by a specific bitcoin address

One of the problems we faced when using the API provided by *Chain.com* was that the free version provided no explicit up-time guarantees. As such, we found that the API could sometimes fail, refusing to accept requests for temporary periods of time. In order to overcome this we needed to make our solution resistant to API failure. In certain scenarios this meant waiting for a specific period of time before making the API request again. In other scenarios however, we opted to forward our request to a backup API, such as those provided by *Chain.so* and *Test.Webbtc.com*. For example, when broadcasting a new transaction to the *Bitcoin* network there is no constraint that says we we have to broadcast that transaction using *Chain.com*. As such, we can make use of the other APIs to perform the same task. When a transaction is broadcast to the network it should ultimately be relayed to every node in the network and so as soon as the *Chain.com* node resumes operation, it too will receive the new transaction. This highlights the distributed nature of the *Bitcoin* network.

Another problem we faced with the *Chain.com* API was that due to its infancy it did not provide support for generating new *OP_RETURN* transactions. Although we could fetch existing *OP_RETURN* transactions from the network, we could not create them using the API. This meant that we needed to manually construct the raw *OP_RETURN* transactions ourselves. To do with we used the *bitcoin-ruby* [116] *Ruby Gem*. This gem provides an implementation of the standard *Bitcoin* protocols and utilities in *Ruby*. Using *bitcoin-ruby* we were able to manually create, manipulate and sign *OP_RETURN* transactions ourselves.

Although using the *bitcoin-ruby* gem to create raw transactions was not too difficult, we did run into several problems regarding the compatibility of internal representations between *bitcoin-ruby* and the *chain-ruby* gems. Internally, both of these gems represent a transaction differently before it is broadcast to the network. As such, this led to us having to manipulate the internal representations of the transactions when we passed them between the gems. For example, in order to create a raw *OP_RETURN* transaction using *bitcoin-ruby*, you need to specify the input transactions to spend. In order to fetch those input transactions from the *Bitcoin* network we used the *chain-ruby* gem to communicate with the *Chain.com* API. Unfortunately, the representation of a transaction fetched by *chain-ruby* is different to the representation expected by *bitcoin-ruby* and so we needed to modify the fetched input transaction before passing to *bitcoin-ruby*. Figure 77 shows a code snippet of part of the transformation we had to perform on the given input transaction. As can be seen in the snippet, we had to manually set the version of the transaction to 1 to maintain backward compatibility with the gem. We also had to modify each of the inputs in the transaction by renaming the keys that were used to look up the values of the previous transaction hash and the signature.

```ruby
# Transforms a transaction object fetched from Chain.com to a transaction object
  that can be used by the 'bitcoin-ruby' gem.
def transform_to_bitcoin_ruby_transaction(input_transaction)
  input_transaction["version"] = 1

  input_transaction["inputs"].each do |input|
    input["previous_transaction_hash"] = input["output_hash"]
    input["script"] = input["script_signature"]
  end

  ...

  return transaction
end
```

Figure 77: Modifying a transaction fetched by *chain-ruby* to a format expected by *bitcoin-ruby*.

In order to monitor payments and settle bets, we needed to find a way to check whether or not a bet has been paid for or an outcome announced. Because this is a continuous process, with different bets being created, paid for and settled all the time, we used *rufus-scheduler* to create a background job that periodically polls the *Bitcoin* APIs. This job runs every few minutes and performs three functions:

1. The first function performed by the job is to update all unpaid bets with their respective payments. We do this by contacting the *Chain.com* API to find any payments that have been made by the user to the unique bet address given to each bet. When a payment is found the bet is updated with the amount received and immediately becomes eligible for matching.

2. The second function performed by the job is bet matching. Any bets that contain unmatched portions are eligible for matching. The pattern we use to match bets is described in the Background, section 2.2.3. This involves matching *back* and *lay* bets on the same outcome together, assuming appropriate odds. In addition we also match *back-back* bets and *lay-lay* bets if the market only has two outcomes. It is worth noting that in order to minimize rounding errors regarding *bet payments*, *bet odds* and *matched portions* we make use of *Ruby BigDecimal* [104].

3. The final function performed by the job is to check whether or not an outcome has been announced for any finished events. For all matched and partially bets we monitor any announcements made into the *Blockchain* regarding the outcome. We do this by using the *Chain.com* API to search for any *OP_RETURN* transactions created by the sports authorities. When an outcome is found, we update and settle the bets in the bet group, paying the winner. Payment for a bet occurs as described in the Research and Design section, 4.2, and uses the private key store. When a payment request is generated we send the request, along with the appropriate proof to the key store. Assuming everything is correct, the key store returns a transaction for the winning payment that can then be broadcast to the network.

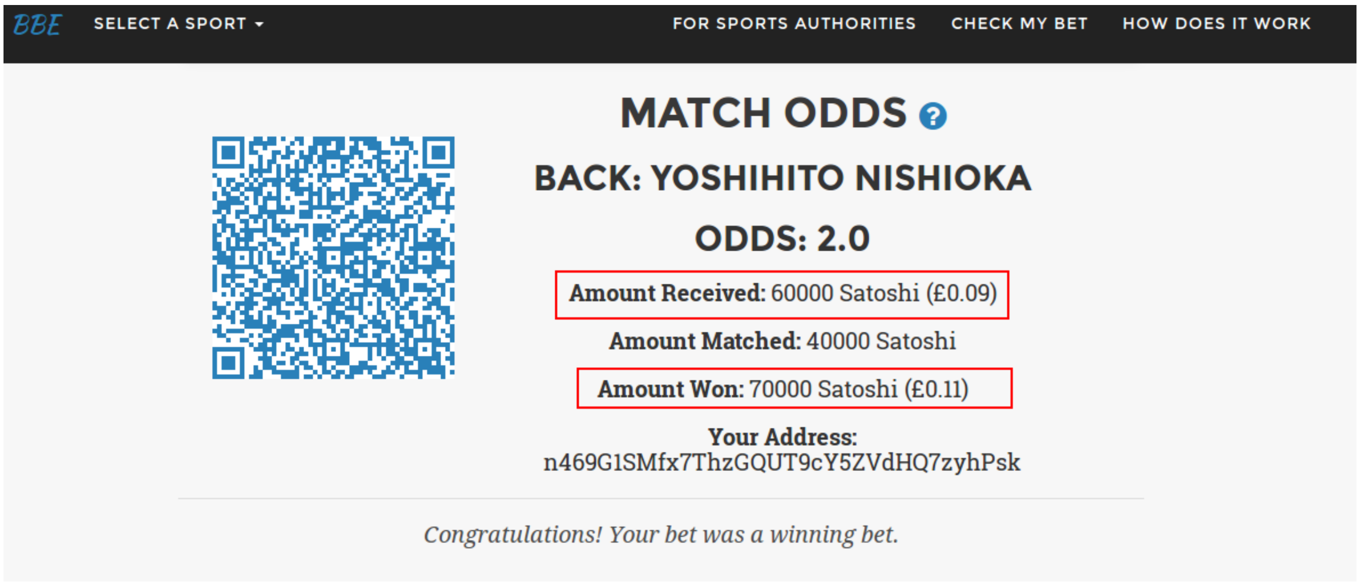
Figure 78 shows a code snippet of the background job. As can be seen, this job is very similar to that of the background job that populates the database with sport and market information. Each of the three functions performed by the job are wrapped in a database transaction. This is to avoid any failures leaving the database in an inconsistent state. Under failure we simply roll-back the transaction, log the error and resume operation. It is also worth noting the "`:mutex => 'database'`" annotation associated with the task. We use this to associate the task with a database mutex, or lock. This locks the database when the task is running, preventing other background tasks from interfering with it during this time. Note that all other requests to the database are still handled as normal and that this does not affect user requests.

```ruby
# Poll the Bitcoin Blockchain for updated payments and settlements every 3
  minutes
scheduler.every '3m', :overlap => false, :mutex => 'database' do
  begin
    ActiveRecord::Base.transaction do
      update_unpaid_bets()
    end
    ActiveRecord::Base.transaction do
      match_bets()
    end
    ActiveRecord::Base.transaction do
      settle_bets()
    end
  rescue => e
    Rails.logger.info "Error when updating bets: #{e.inspect} " + Time.now.to_s
    Rails.logger.info "Back Trace: #{e.backtrace} "
  end
end
```

Figure 78: A background task written using *rufus-scheduler* that periodically updates, matches and settles bets.

6.2.4 Live Financial Data

Whenever a bet is displayed on the *bet status* page we provide real time estimates for the value of that bet in *pounds* (*GBP*). Figure 79 shows a screen-shot of a winning bet. Highlighted in red are the estimated values in *GBP* for the original payment and for the winnings the user received. In order to calculate these estimates and convert between *Bitcoin* and *GBP* we require real-time currency data. To do this we use the *BitcoinAverage* [78] API. *BitcoinAverage* is an aggregated price index for *Bitcoin* that averages data from over 30 different *Bitcoin* exchanges online. It provides many different types of *Bitcoin* price index, including a 24 hour rolling average and the most recent sell and buy prices for *Bitcoin*.



MATCH ODDS

BACK: YOSHIHITO NISHIOKA

ODDS: 2.0

Amount Received: 60000 Satoshi (£0.09)

Amount Matched: 40000 Satoshi

Amount Won: 70000 Satoshi (£0.11)

Your Address:
n469G1SMfx7ThzGQUT9cY5ZVdHQ7zyhPsk

Congratulations! Your bet was a winning bet.

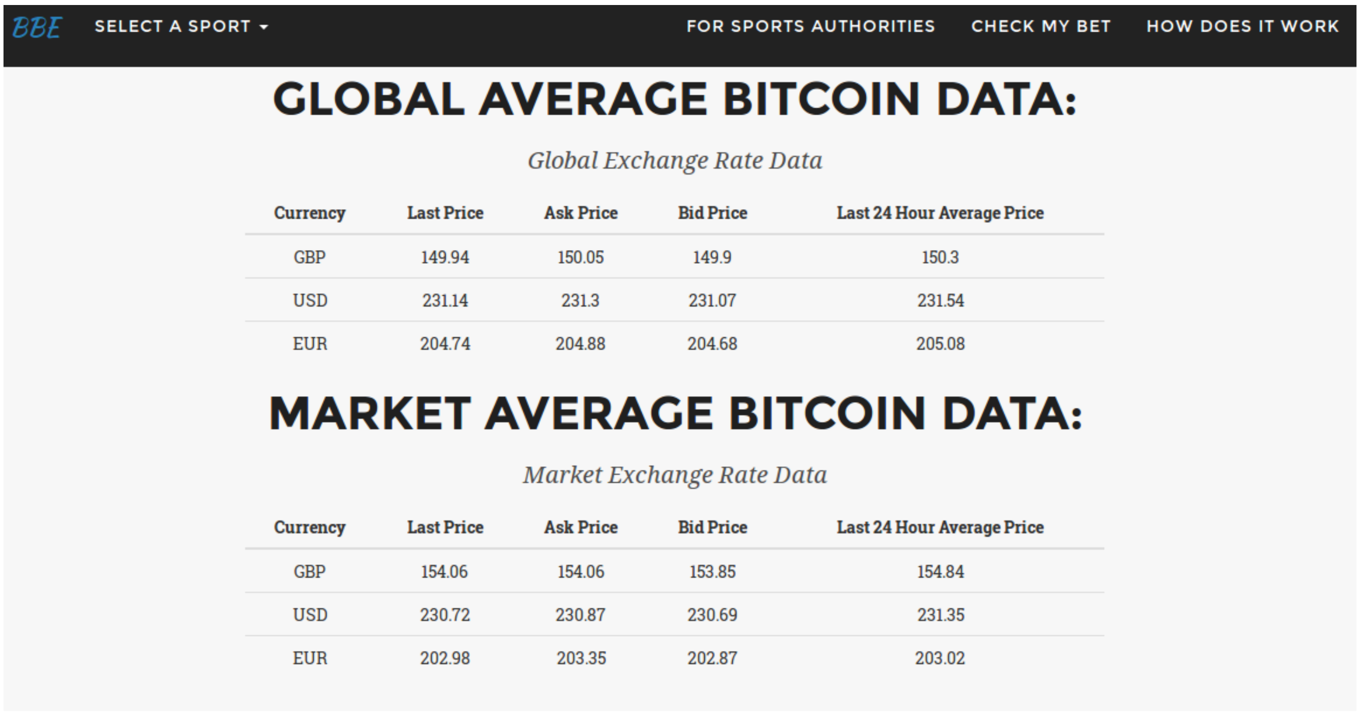
Figure 79: A screen-shot of a winning bet showing the estimated value of the original payment and the received winnings in *GBP*.

To use the *BitcoinAverage* API we make use of the *bitcoinaverage* [17] *Ruby Gem*. This *Ruby Gem* provides a set of methods that make it easy to communicate with the API and keep up to date with the latest prices. Just like our live event and market data, we store the price indexes for *Bitcoin* in our database. This allows quick access to the latest price index without concern of going over the API request limit with many concurrency users. Furthermore, in order to keep this value up to date, we make use of *rufus-scheduler* to run a background task that updates the value every hour. Because this information is only used for display purposes and not for critical financial operations, we felt it would be unnecessary to update the value more often. Figure 80 shows a code snippet for this background task. Note that again we wrap the call to update the database in a transaction to protect against failure. We also write any failures to the server log for debugging later on.

```ruby
# Update BitcoinAverage data every 1 hour
scheduler.every '1h', :overlap => false, :mutex => 'database' do
  begin
    ActiveRecord::Base.transaction do
      BitcoinAverageDatumPopulator.fetch_and_populate
    end
  rescue => e
    Rails.logger.info "Error in BitcoinAverage populator: #{e.inspect} " + Time.
  now.to_s
    Rails.logger.info "Back Trace: #{e.backtrace} "
  end
end
```

Figure 80: A scheduled job written using *rufus-scheduler* to update the database with new *BitcoinAverage* data every hour.

Figure 81 shows a screen-shot of the various types of *BitcoinAverage* data we store in the database. As can be seen in the screen-shot we store the latest *ask* and *bid* prices for *Bitcoin*, as well as the 24 hour rolling average.



GLOBAL AVERAGE BITCOIN DATA:

Global Exchange Rate Data

Currency	Last Price	Ask Price	Bid Price	Last 24 Hour Average Price
GBP	149.94	150.05	149.9	150.3
USD	231.14	231.3	231.07	231.54
EUR	204.74	204.88	204.68	205.08

MARKET AVERAGE BITCOIN DATA:

Market Exchange Rate Data

Currency	Last Price	Ask Price	Bid Price	Last 24 Hour Average Price
GBP	154.06	154.06	153.85	154.84
USD	230.72	230.87	230.69	231.35
EUR	202.98	203.35	202.87	203.02

Figure 81: A screen-shot of the *BitcoinAverage* data stored in the database.

6.2.5 Testing

Given that our application operates using the *Bitcoin* currency and is designed to handle finances, it was essential that we thoroughly tested all of the critical aspects of our code. This is because a single mistake in a calculation could result in an incorrect payment or incorrect bet match. Given the delicate nature of *Bitcoin* (e.g. the fact that any money that is incorrectly sent to the wrong address cannot be re-claimed) this could have devastating consequences for us and our users.

One aspect that provided relief from these concerns was the fact that our private key store validates bet information and payments before signing transactions. This means that if a bug did exist in our implementation and we incorrectly paid a user, it would not go unnoticed by the key store. Nevertheless, it was still important to thoroughly test the critical aspects of our code.

In order to do this we made use of *RSpec* [102], a very popular *behaviour driven development* testing framework for *Ruby*. By default *Rails* applications come with the *rspec-rails* [103] gem installed and as such provide a very easy way to begin testing *Rails* applications. Using *RSpec* we can write many different types of tests for our application, such as *unit* tests, *end-to-end* tests and *integration* tests. Given that we were taking an *agile* and *test driven development* approach to our application, where tests are written before features are implemented, *RSpec* was incredibly useful during development.

In addition, *RSpec* supports *mockist* testing, providing easy ways to mock various objects that the *class under test* communicates with. This was especially useful when testing the communication and interaction between the classes in our application and external APIs. For example, when writing the code to poll the *Bitcoin* network to check whether or not a user has paid for their bet, it is much more appropriate to *mock* the API we are using than to use the real API in our tests. This is because, firstly we don't want to be making real API calls in our tests, as this slows down the test infrastructure and requires real data to exist. Secondly, API calls can also fail too, and so we don't want multiple failing tests to show a feature not working when it is in fact the API that has failed. Instead, we can *mock* the *API* and specify exactly what calls we expect to be made on it and what those calls should return. This makes it much easier to test communication with the external APIs.

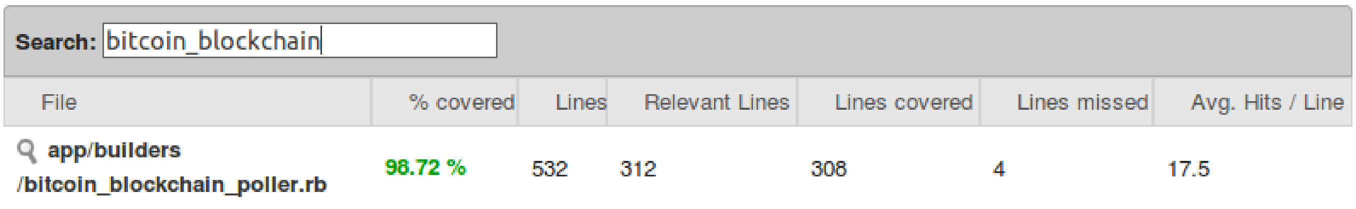
For example, figure 82 shows a code snippet of a test for the `BitcoinBlockchainPoller`. The `BitcoinBlockchainPoller` is the class that polls the *Bitcoin* network in order to update, match and settle bets. This is described in section 6.2.3 above. As can be seen in the code snippet, we have a single test that checks that the poller correctly updates the status of a bet that has been paid for by the user. It does this by *mocking* the *Chain.com* API client (`@mock_chain_client`) and telling it to expect a call to get the unspent transactions for a given bet address. In addition, we also tell the mock that when it receives this call it should return a single transaction. The *object under test*, in this case the `BitcoinBlockchainPoller`, then updates the unpaid bets and we check that the bet has been appropriately updated with the users payment. This example highlights the benefits of using *mockist* testing.

```
describe BitcoinBlockchainPoller do
  ...
  it "should update the status of the bets that receive payments" do
    unpaid_bet = create_unpaid_bet(TEST_BITCOIN_ADDRESS_1, TEST_BITCOIN_ODDS_1)
    transaction = create_new_transaction(TEST_TRANSACTION_AMOUNT)
    expect(@mock_chain_client).to
      receive(:get_address_unspents).with(TEST_BITCOIN_ADDRESS_1).and_return(
  transaction)

    @bitcoin_blockchain_poller.update_unpaid_bets

    expect(unpaid_bet.status).to eq(STATUS_PAYMENT_RECEIVED)
    expect(unpaid_bet.payment_received).to eq(TEST_TRANSACTION_AMOUNT)
  end
  ...
end
```

Figure 82: A code snippet of one of our tests that checks whether or not the Bitcoin Blockchain Poller correctly updates bets that have been paid for. This test *mocks* the API we are using.

In order to help guide us when testing our code and protect us from missing important behaviours, we added the *simplecov* [89] *Ruby Gem* to our test framework. The *simplecov* gem provides code coverage metrics for *Rails* applications. It highlights areas that need improvement, such as those with a very low test coverage and also provides a line-by-line view of all files, showing you exactly which lines have been covered and which haven't. Although code coverage does not provide an absolute indicator of how well a file is tested it does give us some idea about the areas that need improvement. As such we used *simplecov* to help guide us when writing tests for the most critical areas of our code.

For example, figure 83 shows a screen-shot of the output produced by *simplecov*. As can be seen in the screen-shot, the `BitcoinBlockchainPoller` module has a code coverage of over 98%. Given that the `BitcoinBlockchainPoller` is quite a crucial aspect of our application, as it is responsible for updating, matching and settling bets, it is important to have adequate test coverage for this part of the application. Furthermore, we can use *simplecov* to look up exactly which lines were missed in order to see how we can improve our tests. Figure 84 shows a screen-shot of a line that was missed when testing the `BitcoinBlockchainPoller`. This is highlighted in red. In order to get to this line, you would need to be in a state that after fetching the transactions for a specific bet address from the network, one of the transactions you fetched is not for that address. This scenario is incredibly unlikely given the expected behaviour of the API. As such, assuming there are no bugs in the API, it could be considered a false alarm. If however, you were pushing for 100% test coverage, you would create a test runs through this exact scenario. This highlights the benefits of using *simplecov*.


Search: bitcoin_blockchain

File	% covered	Lines	Relevant Lines	Lines covered	Lines missed	Avg. Hits / Line
app/builders /bitcoin_blockchain_poller.rb	98.72 %	532	312	308	4	17.5

Figure 83: A screen-shot of the output produced by *simplecov*. This shows us that the `BitcoinBlockchainPoller` module has coverage of over 98%, with only 4 lines missed.

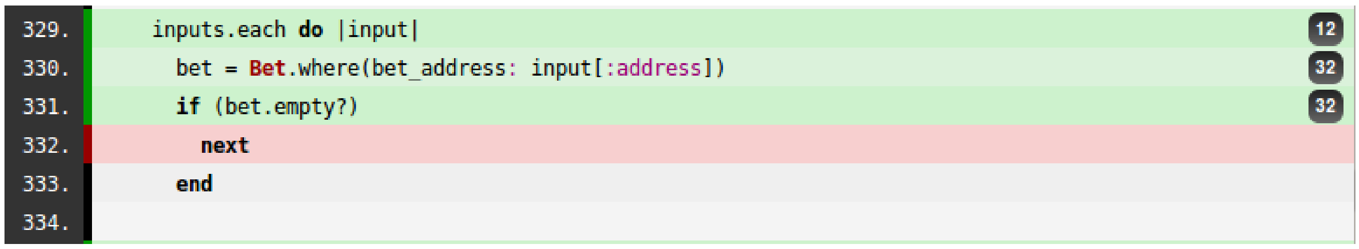

```
  inputs.each do |input|                                12
    bet = Bet.where(bet_address: input[:address])       32
    if (bet.empty?)                                     32
      next
    end

```

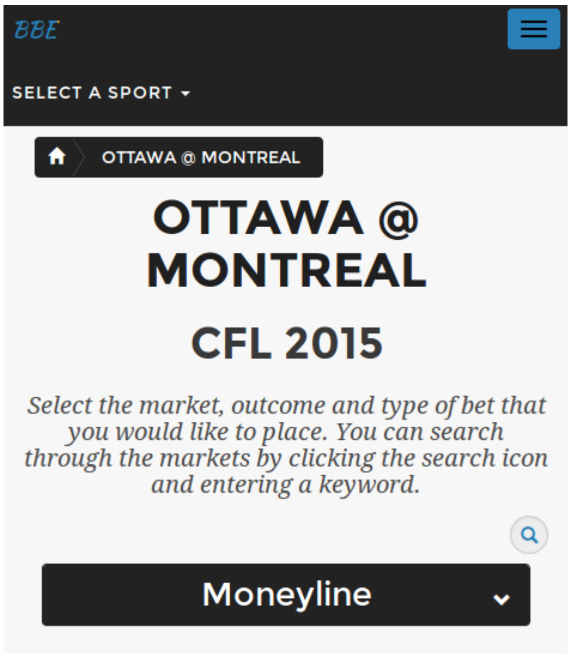
Figure 84: A screen-shot of the output produced by *simplecov*. This shows us one of the lines not covered by our `BitcoinBlockchainPoller` tests.

6.2.6 User Interface and Experience

Twitter Bootstrap

When designing and building the user interface for our web application, we made use of *Twitter Bootstrap* [114] for the majority of our layouts and components. *Twitter Bootstrap* is a free and open-source collection of components that can be used for web applications and design. It is very commonly used in scenarios that require rapid prototyping and fast iteration, as it provides a standard set of commonly used web components without having to build them from scratch. We used *Bootstrap* for many of the components on our web application, for example, the buttons, grid system and navigation bar. Although *Twitter Bootstrap* provides a standard styling for all of its features, we added a large amount of customization to our interface. This was to avoid our website appearing identical to many of the generic web applications that also use *Bootstrap*.

One of the great things about *Bootstrap* is that it is responsive by design, meaning that by default it has been designed with mobiles and tablets in mind. This is useful for our application as it allows users to view the website on both mobile and tablet devices, as well as larger screens. Figure 85 shows a screen-shot of the *Bitcoin Betting Exchange* as it would appear on a smaller screen, such as a tablet. As can be seen in the screen-shot, the navigation bar at the top of the site collapses and the breadcrumbs shorten to fit on the smaller screen.

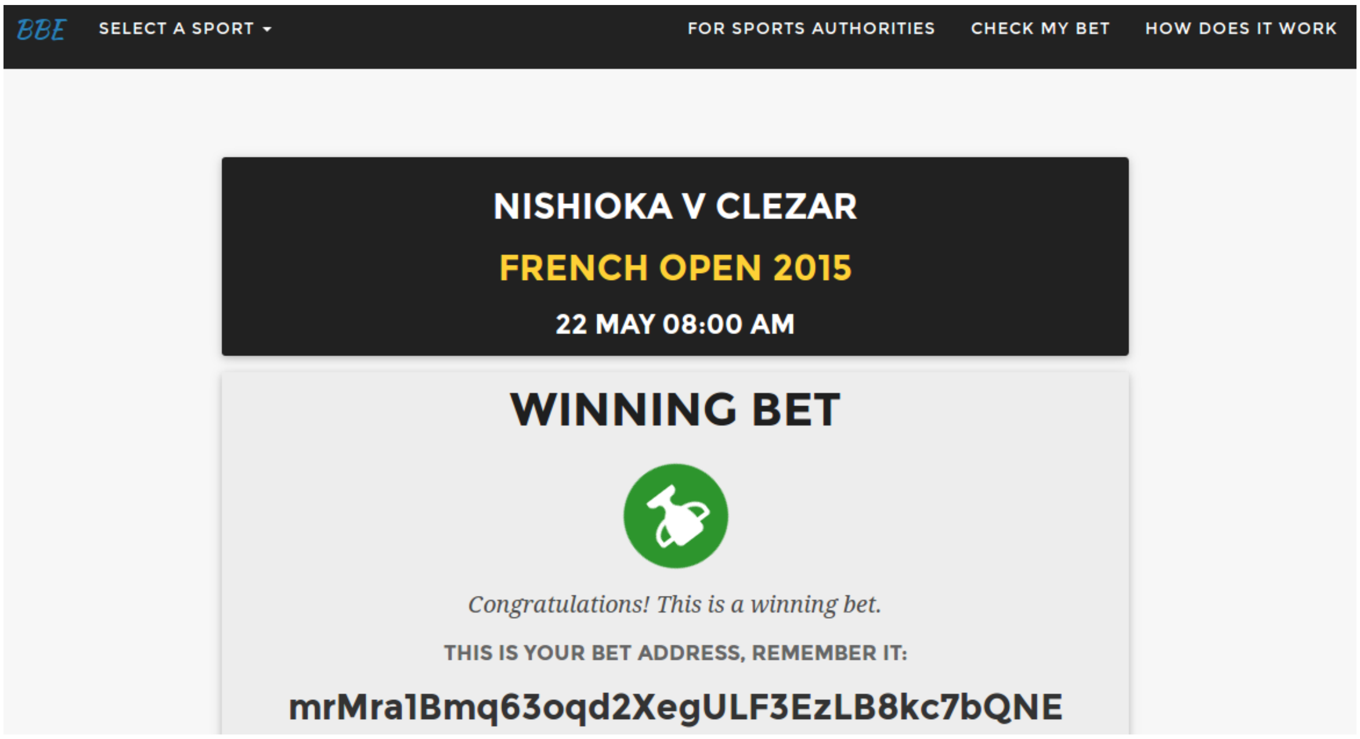


Select the market, outcome and type of bet that you would like to place. You can search through the markets by clicking the search icon and entering a keyword.

Figure 85: A screen-shot of the *Bitcoin Betting Exchange* that highlights its responsible design.

AJAX

In order to improve the overall user experience of the website, we also added several useful features to make it more responsive, interactive and intuitive for the user. The first of these features is implementing *AJAX* [99] calls to our bet pages. As previously mentioned when a user visits their *bet status* page, they are presented with all of the information relating to their bet. As this information might update according to various actions, we use *AJAX* to automatically reload part of the page in background. For example, when a user has paid for their bet and their bet status is updated, they do not need to manually refresh the page as it automatically reloads for them. Figure 86 shows a screen-shot of the *bet status* page that automatically reloads without interfering with the user.



NISHIOKA V CLEZAR

FRENCH OPEN 2015

22 MAY 08:00 AM

WINNING BET

Congratulations! This is a winning bet.

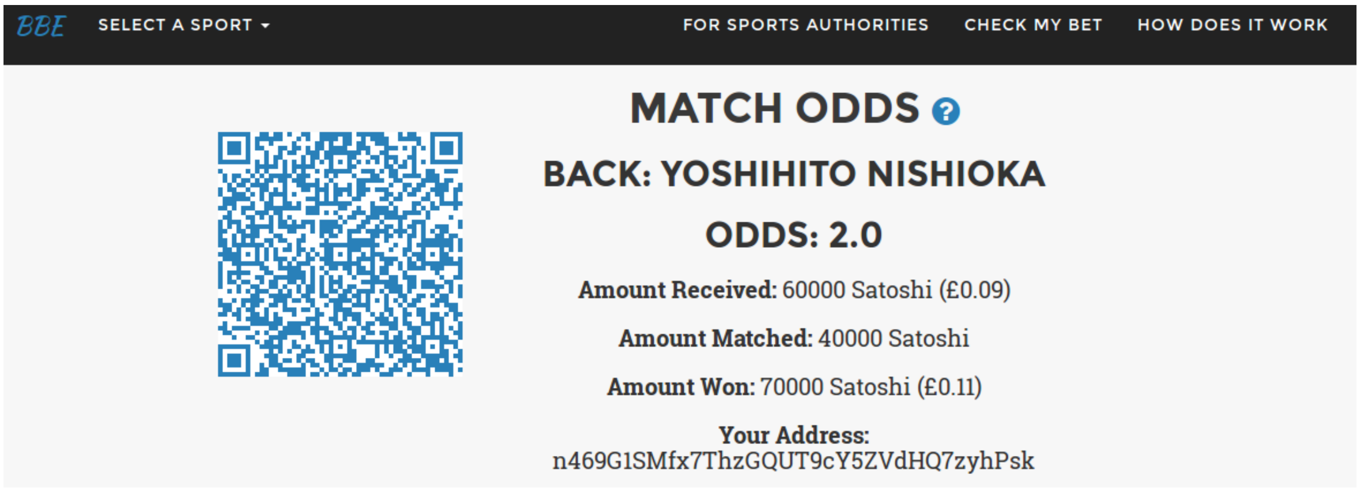
THIS IS YOUR BET ADDRESS, REMEMBER IT:

mrMra1Bmq63oqd2XegULF3EzLB8kc7bQNE

Figure 86: A screen-shot of the *Bet Status* page that uses *ajax* to automatically reload whenever it is updated.

QR Code Generation

To make it easier for users to pay for their bets we provide a unique *QR code* for each bet. This *QR code* uses *Bitcoin Improvement Protocol 21* (*BIP0021*) [117] and the *rqrcode-with-patches* [28] *Ruby Gem* to generate a unique *URI* so that it can be scanned by any *Bitcoin* wallet. Providing a *QR code* is much safer than expecting users to enter the address into their wallet manually. This is because *Bitcoin* addresses are not very user friendly and hence prone to human error. As such a mistake would result in the money being sent to the wrong address and would be permanently lost forever. Instead a user can simply scan the code with their wallet and a payment will be automatically generated by that wallet. This provides a much more user friendly experience when paying for bets. Figure 87 shows a screen-shot of a unique *QR code* generated for a bet.



MATCH ODDS

BACK: YOSHIHITO NISHIOKA

ODDS: 2.0

Amount Received: 60000 Satoshi (£0.09)

Amount Matched: 40000 Satoshi

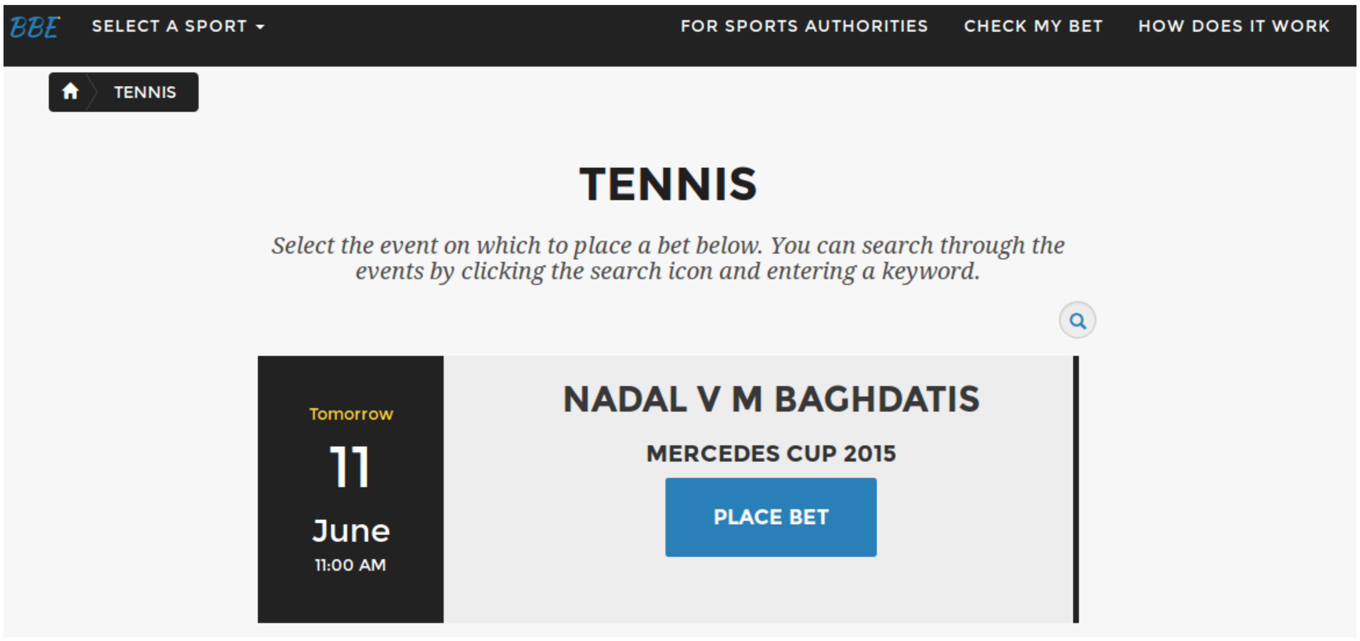
Amount Won: 70000 Satoshi (£0.11)

Your Address:
n469G1SMfx7ThzGQUT9cY5ZVdHQ7zyhPsk

Figure 87: A screen-shot of a *QR code* generated for a bet.

Events Displayed in Local Time

Another improvement we made to our user interface was to display the starting time and date of each event in the user's local time. This allows each user to have a customized view of the events page that depends on exactly where in the world they are located and what time zone their browser has been set to. We do this using the *local_time* [8] *Ruby Gem*. This allows users to know exactly when each event begins relative to them, regardless of whether or not the event is being held in their local time zone. Figure 88 shows a screen-shot of the *Tennis* events currently available, displayed in *Greenwich Mean Time*.

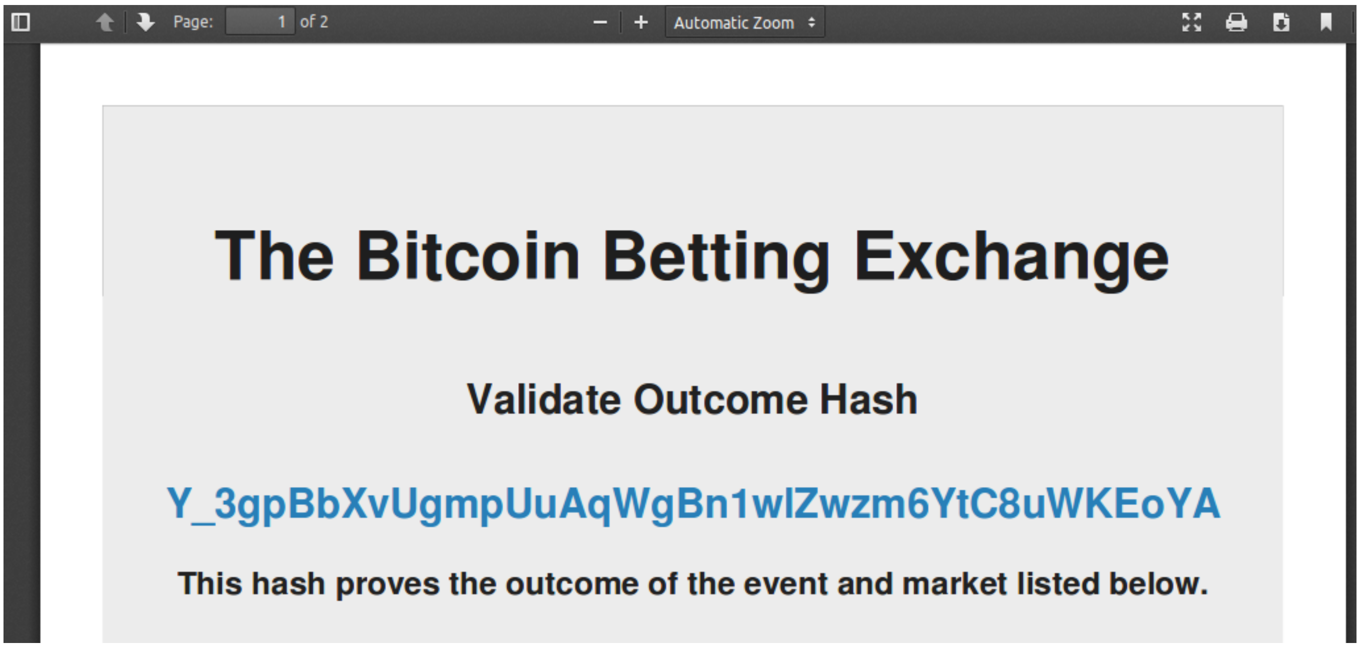


Select the event on which to place a bet below. You can search through the events by clicking the search icon and entering a keyword.

Figure 88: A screen-shot of the *Tennis* events displayed in *Greenwich Mean Time*.

PDF Generation

Another improvement we made to our website was to provide downloadable *PDFs* of *bet* and *outcome* proofs. Instead of requiring the user to save the proof by hand (e.g. copy the proof into a text document or save the web page locally), we allow them to download a clearly laid out and well-explained copy of their proof in *PDF* format. To provide these *PDFs* we make use of the *wisepdf* [3] and *wkhtmltopdf-binary* [130] *Ruby Gems*. These gems enable us to automatically generate *PDFs* of various templates as required. A *PDF* download, as opposed to a confirmation email, maximises user anonymity. Figure 89 shows a screen-shot of a *PDF* generated for an *outcome*.



The Bitcoin Betting Exchange

Validate Outcome Hash

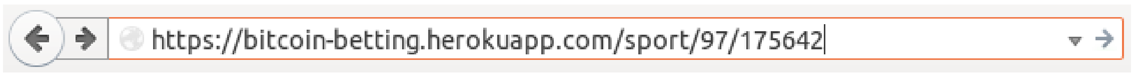
Y_3gpBbXvUgmpUuAqWgBn1wlZwzm6YtC8uWKEoYA

This hash proves the outcome of the event and market listed below.

Figure 89: A screen-shot of a *PDF* generated for an *outcome* proof.

Attractive URLs

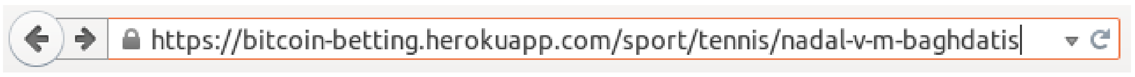
The final change we made to our website to improve the overall user experience was to implement attractive urls. By default *Rails* embeds the unique id of each database object into the url when it is viewed by the user. This produces urls that not very user friendly and difficult to understand. Figure 90 shows an example of a url produced for a tennis event.



https://bitcoin-betting.herokuapp.com/sport/97/175642

Figure 90: A screen-shot of the default url provided by *Rails*. This url embeds the unique ids for each object.

In order to overcome this we use the *friendly_id* [37] *Ruby Gem*. This gem allows us to substitute user friendly strings for unique ids in urls. This produces *permalink* urls that are easier to understand and improves the user experience. Figure 91 shows the same url as that in figure 90 but this time using custom strings. As can now be seen in this url, the sports event is a *tennis* event for the match of *Nadal vs. Baghdatis*.



https://bitcoin-betting.herokuapp.com/sport/tennis/nadal-v-m-baghdatis

Figure 91: A screen-shot of a much more user friendly url for the *Tennis* event of *Nadal vs. Baghdatis*.

In addition to string based urls being more user-friendly, they also help prevent *implementation leakage*. Although knowing the id of a database object is not a vulnerability in itself, it does allow an attacker to infer information about how the application is built and so it may aid them in finding weaknesses. For example, *DirectBet* currently suffer from this problem. Their urls contain the unique ids of their database objects and so by poking around and comparing these ids to the ones provided by *Betfair* for the same events, you can find that the id's are identical, telling us that *DirectBet* get their data from *Betfair*. This means that if an attacker wanted to attack them in some way, a possible place to start would be to monitor the communication between *DirectBet* and *Betfair*. Figure 92 shows the url displayed by *DirectBet* for a *tennis* event.

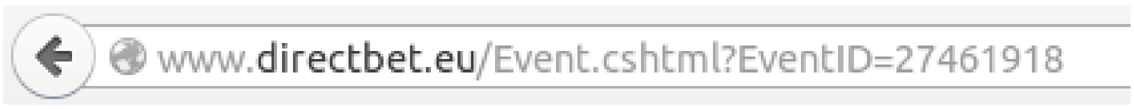

www.directbet.eu/Event.cshtml?EventID=27461918

Figure 92: A screen-shot of the url displayed by *DirectBet* for a tennis event.

6.3 The Smart Private Key Store

6.3.1 Infastructure & Platform

When deciding on what tools and platforms to use to build our software implementation of the *smart private key store*, it was clear that in order to make the key store as secure as possible, it would be wise to use as minimal a server as possible. This is because the more features, layers, and parts you add to the server, the more potential there is for vulnerabilities to arise. As is often stated, *"complexity is the enemy of security"*, and this is especially true when operating a web application that can be accessed by anyone online.

Because our software implementation does not have the benefits that a hardware implementation has, there will nonetheless be issues present in our implementation that will not be a concern for the hardware device. For example, in order to communicate with a hardware device connected to an external server, you would either need to have physical access to that device or find a vulnerability in the server that allows to you to communicate with it. This therefore adds some protection to the hardware key store that is not present in an online web application.

Because of the numerous benefits that *Ruby on Rails* provides, such as its many external libraries, the ability to host it online easily at low cost and its large support community, we thought it would be appropriate to implement our key store using *Ruby on Rails*. Rather than using a normal *Rails* application which has many separate and moving parts, we opted instead to use a much simpler, stripped-down, API-only version of *Rails*. To do this we used the *rails-api* [95] *Ruby Gem*. This gem transforms a normal *Rails* application into one that is a subset of the original by completely removing many of the unnecessary and unneeded components from the application. This produces a much simpler and lightweight application. Considering that our key store does not need to provide any user interface, templates or content, an API-only application is perfect for our needs. This is because communication between the betting exchange and the key store can be modelled using an API.

Just like our betting exchange web application, the key store will make use of a database to mimic local storage. As mentioned, all communication between the application and its database is encrypted, preventing an attacker from simply listening to the information that flows between them.

6.3.2 API

As described in the Research and Design chapter, section 4.2, our key store only needs to provide 5 API methods to satisfy the needs of the betting exchange. These are:

1. Generate a new *Bitcoin* address and signed bet proof for a specific bet.
2. Update the payment received for a specific bet. This is the payment sent by the user.
3. Update the amount matched for a specific bet.
4. Return the unmatched portion of a specific bet. This is the remaining amount of the user's payment not yet matched when the event began.
5. Pay the user for their winning bet.

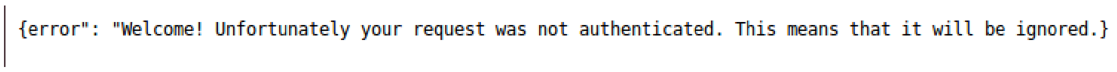
In order to keep the representation of *Bitcoin* transactions consistent across the betting exchange, the APIs and the private key store, we decided to implement our API as a *RESTful JSON* API[96]. *REST* typically stands for *"REpresentational State Transfer"* and is a state-less client server protocol. The advantages of using a *RESTful* API are that it requires no state to be stored by the server regarding the client. Instead each client request is self contained and uniquely identifies a resource it requires. This reduces the coupling between the server and the client and improves such as scalability, simplicity and performance. This therefore reduces the complexity of the key store and makes it easier to model and reason about. In addition, we decided to use *JSON* (*Javascript Object Notation*) as our data representation because *Bitcoin* transactions are already represented as *JSON* when sent between the betting exchange and the various *Bitcoin* APIs we use. This has that added benefit that we can keep communication between all parties consistent and so again, reduce the complexity of our design.

In order to implement this API we configured 6 specific routes in our *Rails* application. Each route represents a unique API method, with an additional route for the homepage of the key store, which is just for demonstration purposes. By explicitly adding these routes to the *Rails* application, all other paths and requests are ignore, meaning we only need to worry about securing these paths. Figure 93 shows a code snippet of our application's *routes* file. As can be seen there are 6 specific routes, the first of which is the default route that handles all requests to the homepage, and the other 5 are for each of the API methods in turn. Note that we explicitly limit the type of *HTTP method* that can be performed on each path (e.g. *GET* or *POST*). This means that all other *HTTP method* requests to these paths are ignored. Again, this simplifies the number of routes we need to secure. All API requests are routed through the path *".../api/{REQUEST}"* making it easy to see exactly which request is being made.

```
Rails.application.routes.draw do
  # HOMEPAGE
  root ...

  namespace :api, defaults: {format: 'json'} do
    # GET /api/get_new_bet_address
    get 'get_new_bet_address/:return_address' ...

    # POST /api/payment_received
    post 'payment_received' ...

    # POST /api/update_matched
    post 'update_matched' ...

    # POST /api/sign_bet_return
    post 'sign_bet_return' ...

    # POST /api/settle_bets
    post 'settle_bets' ...
  end
end
```

Figure 93: A code snippet of our application's *routes* file. This file only exposes 6 routes in the key store API.

We implement each method exactly as described in section 4.2, performing the necessary checks on each API call before constructing the appropriate response. Similarly to the betting exchange we use the *bitcoin-ruby* and *chain-ruby* gems to perform transaction validation and signing locally. This means that that private keys are never exposed to the outside world and that our key store never has to communicate with any external APIs. Whenever a request is made to our API, we require the arguments of the request to be in *JSON* and appropriately formatted. Likewise each of our responses returns JSON and a status code depending on the response. Figure 94 shows a screen-shot of the homepage of the key store when visited through a browser. This request returns a *401* unauthorized status code and an error message rendered in JSON. This is because the request made by the browser was not authenticated and so the request was ignored.



```
{error": "Welcome! Unfortunately your request was not authenticated. This means that it will be ignored.}
```

Figure 94: A screen-shot of the default route of the private key store when accessed through a browser. This returns a *401* unauthorized status code and displays an error because the request has not been authenticated.

One benefit of targeting simplicity in our design is that we only need to define 6 methods in our application, one for each API call. This allows us to pay careful attention to each method and to avoid introducing vulnerabilities into our application. For example the implementation of the method that handles requests to the homepage can be seen in figure 95. The `request` object is passed directly to the method when the API call is made. If the `request` cannot be authenticated the API call is refused. This highlights the simplicity of our implementation.

```
module Api
  ...
    def home
      if (!authenticate_request(request))
        render json: {"error": "Welcome! Unfortunately your request was not
    authenticated. This means that it will be ignored."}, status: 401
      else
        render json: {"message": "Welcome! Your request was successfully
    authenticated!"}, status: 200
      end
    end
  ...
end
```

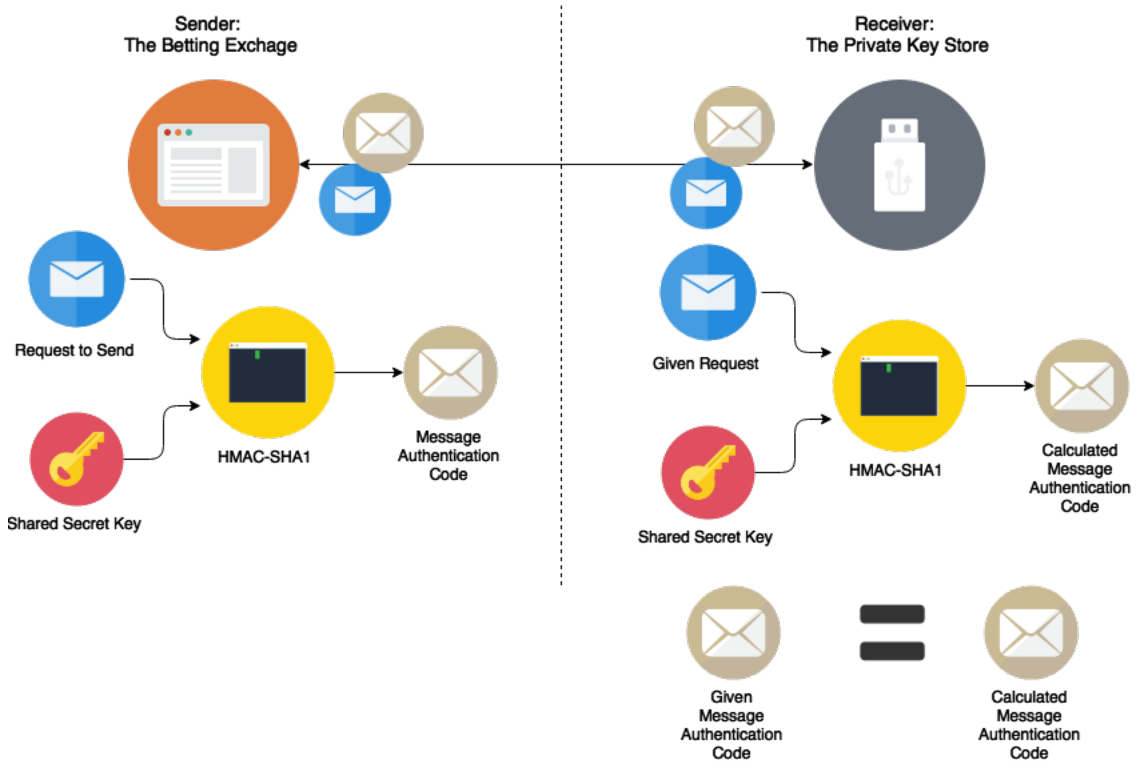
Figure 95: A code snippet of the method that handles requests to the homepage of our wallet.

6.3.3 Authentication

To prevent anyone from being able to communicate with the private key store, we require all requests to be authenticated. As described in section 4.2, this can be done by *bootstrapping* a shared secret or key to the hardware device and sharing this key with the betting exchange. In this case any requests made from the exchange to the device will have to be authenticated using the shared secret. If authentication fails, the private key store will treat the request as coming from someone other than the exchange, and simply refuse it. This requires all requests to be authenticated using the shared secret.

To perform authentication of requests we use the *api_auth* [54] *Ruby Gem*. The *api_auth* gem provides support for adding *message authentication codes* to API requests. It does this by using *HMAC-SHA1*, a *keyed-hash message authentication code* that allows us to verify both *authentication* of requests (e.g. that the request came from the betting exchange) as well as *integrity* of requests (e.g. that the request was not changed along the way). *HMAC-SHA1* is quite well known, generally understood and currently considered safe (i.e. it has not yet been broken). For example, *Amazon Web Services* currently use it as their client-server API authentication mechanism. As such, we felt it appropriate to use *HMAC-SHA1* in our solution.

Figure 96 shows how *HMAC-SHA1* works. In the figure we can see that the betting exchange wants to send a request to the key store. In order to produce a *keyed-hash message authentication code* for the request, it takes the request and the shared secret and runs them through the *HMAC-SHA1* algorithm to produce a valid *MAC* tag for the request. The betting exchange then sends the original request as well as the *MAC* tag to the key store. In order to verify *authentication* and *integrity* of the request, the private key store takes the given message and calculates the *MAC* tag itself. It then compares the calculated tag to the one given by the betting exchange. If these are identical it knows that the request was generated by the exchange and that it was not modified along the way.



Sender:
The Betting Exchage
Receiver:
The Private Key Store
Request to Send
Shared Secret Key
HMAC-SHA1
Message Authentication Code
Given Request
Shared Secret Key
HMAC-SHA1
Calculated Message Authentication Code
Given Message Authentication Code
Calculated Message Authentication Code

Figure 96: A diagram illustrating how a *keyed-hash message authentication code* is calculated for a specific API request.

The *api_auth Ruby Gem* does exactly this. It generates these *MAC* tags for every request that the betting exchange sends to the key store. In addition, in order to avoid *replay-attacks* (where the same message is sent to the key store again by an attacker) *api_auth* implements a time-out mechanism where requests expire automatically after a specific amount of time. Similarly *api_auth* also provide several useful methods to generate strong shared secrets, preventing our applications from authenticating using secrets that are weak and vulnerable to attack.

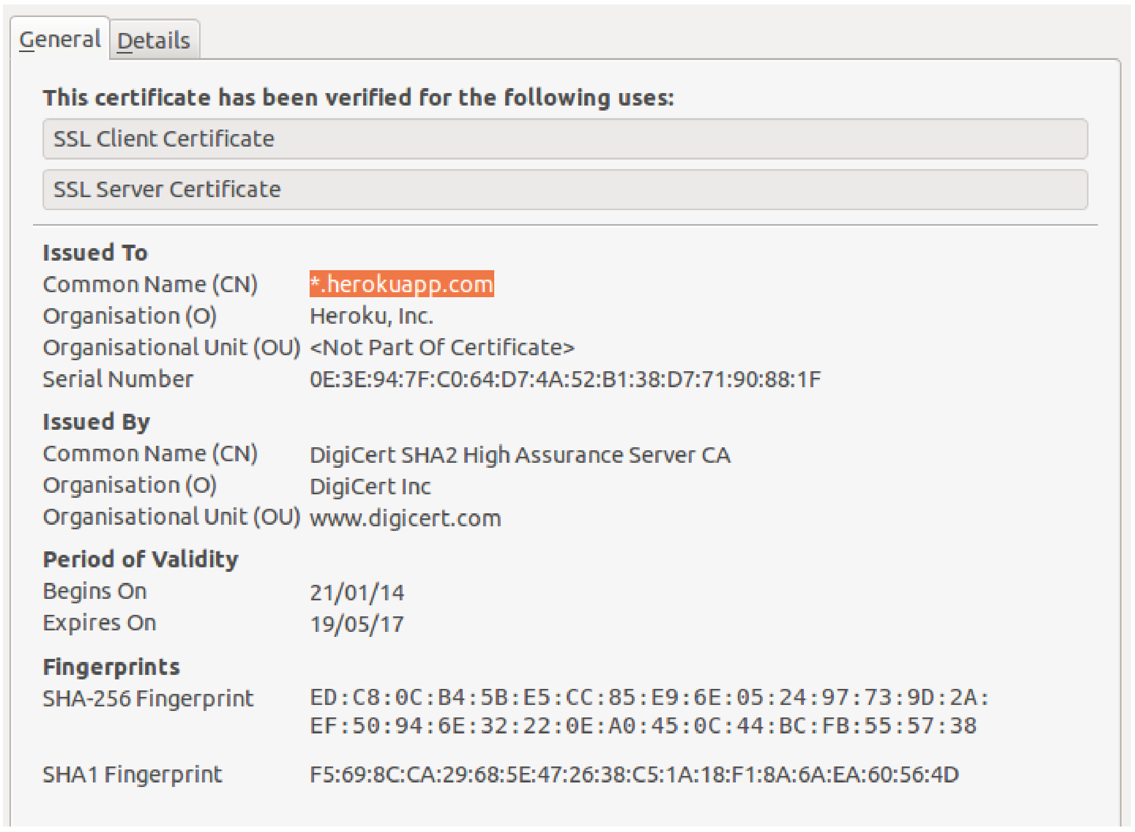
Figure 97 shows a code snippet of our implementation of the method that authenticates all given requests. This was seen previously when we looked at the implementation of the method that handles requests to the default route of our key store. As can be seen in the figure 97, we require all API requests to include an `ACCESS_ID`. This is just another shared secret required between the two applications, and hence adds another layer of security to communication. This means that all API requests need to present this shared `ACCESS_ID` in the request as well as authenticate the request using the shared secret key. The method therefore checks that the `ACCESS_ID` is correct before verifying the *HMAC-SHA1 MAC* tag of the request using the `SHARED_SECRET_KEY`.

```
  ...
    def authenticate_request(request)
      given_access_id = ApiAuth.access_id(request)

      if (given_access_id != SECRET_ACCESS_ID)
        return false
      end

      return ApiAuth.authentic?(request, SHARED_SECRET_KEY)
    end
  ...
```

Figure 97: A code snippet of the method that handles authentication of API requests.

6.3.4 Communication with the Betting Exchange

As mentioned, all communication between the betting exchange, private key store, external APIs and databases are encrypted using either *HTTPS* or *SSL*. This means that whenever an API call is made to the key store from the exchange, it does so over *HTTPS* and therefore an attacker cannot monitor the parameters and API calls being made by the exchange. To implement *HTTPS* in our *Rails* applications we *piggyback* [60] on the *"*.herokuapp.com SSL certificate"* provided by *Heroku* and force our applications to respond only to *HTTPS* requests. This allows us to provide an encrypted communication for our users, as well as between our services. Note that in a commercial setting it would be more appropriate to purchase our own *domain names* and *SSL certificates*. This would mean that users would not have to trust the *SSL certificate* provided by *Heroku*, but instead trust only ours. Figure 98 shows a screen-shot of the *SSL certificate* provided by *Heroku* for our applications.



This certificate has been verified for the following uses:

SSL Client Certificate

SSL Server Certificate

Issued To

Common Name (CN)	*.herokuapp.com
Organisation (O)	Heroku, Inc.
Organisational Unit (OU)	<Not Part Of Certificate>
Serial Number	0E:3E:94:7F:C0:64:D7:4A:52:B1:38:D7:71:90:88:1F

Issued By

Common Name (CN)	DigiCert SHA2 High Assurance Server CA
Organisation (O)	DigiCert Inc
Organisational Unit (OU)	www.digicert.com

Period of Validity

Begins On	21/01/14
Expires On	19/05/17

Fingerprints

SHA-256 Fingerprint	ED:C8:0C:B4:5B:E5:CC:85:E9:6E:05:24:97:73:9D:2A: EF:50:94:6E:32:22:0E:A0:45:0C:44:BC:FB:55:57:38
SHA1 Fingerprint	F5:69:8C:CA:29:68:5E:47:26:38:C5:1A:18:F1:8A:6A:EA:60:56:4D

Figure 98: A screen-shot of the *HTTPS* certificate provided by *Heroku* for our web applications.

In addition to *HTTPS* we also encrypt all of the parameters passed between the exchange and the key store using *RSA 2048 bit encryption*. This is because some of the information we pass between the applications is highly sensitive. If this information was leaked due to a bug in *Rails' SSL* implementation, it would open our application up to many types of attack. Although a bug in the standard *OpenSSL Ruby* library should not exist, with the announcement of the *HeartBleed* bug only a few months ago, it is better to err on the side of caution. In addition, because our API does not need to handle an incredibly large load, the extra time required to encrypt and decrypt this information is not a concern.

Figure 99 shows a code snippet of the API method that updates the amount matched for a specific bet. Assuming the request has successfully authenticated, we decrypt the given `bet_address` and `total_matched` parameters using the `decrypt_api_parameter` method. We then validate that the given parameters are correct (e.g. we will check here that the bet address represents a valid bet and that the total matched is greater than what has been previously matched). The method that decrypts the given parameter can be seen in figure 100. Note that in order to construct the *RSA* private key to decrypt the given string we require the *RSA* private key file as well as the password.

```
# Update the amount matched for a specific bet
def update_matched
  ...
    bet_address = decrypt_api_parameter(params[:bet_address])
    total_matched = decrypt_api_parameter(params[:total_matched])
    if (!is_valid_bet_address(bet_address) || !is_valid_total_matched(
  total_matched, bet_address))
      render json: {"error": "Invalid Arguments Given"}, status: 400
    else
      ...
    end
  ...
end
```

Figure 99: A code snippet of the API method that updates the amount matched for a bet.

```
# Decrypt the given parameter
def decrypt_api_parameter(encrypted_string)
  private_key = OpenSSL::PKey::RSA.new(File.read(PRIVATE_KEY_FILE),
  PRIVATE_KEY_PASSWORD)
  return private_key.private_decrypt(encrypted_string)
end
```

Figure 100: A code snippet of the method that decrypts a given API parameter using an *RSA 2048 bit* private key.

6.3.5 Database Encryption

In order to prevent an attacker from reading the private keys directly off the hardware private key store, we suggested encrypting those keys before storing them on the device. The *decryption* key can then be passed to the device when needed and discarded after use. This prevents an attacker from simply picking up the device and reading the information held on it. The same can be argued for our software implementation. Storing private keys as plain-text in the database of our application is not wise. This is because if an attacker could connect to the database they could extract all of the private keys out of it. Instead we store the private keys in the database under encryption, with the key to decrypt the information held externally to the application. We do this using using the *strongbox* [64] *Ruby Gem*.

Strongbox provides a public key encryption mechanism for *Active Record*. It uses *RSA 2048 bit* encryption to encrypt the data before writing it to the database. Because it uses public key encryption, it means that the key store can write data to the database without requiring the *decryption* key. The *decryption* key is only required when the key store needs to read and decrypt information from the database. As such, we hold the *decryption* key in the betting exchange and pass it directly to the key store when the key store needs to sign a transaction. This means that the database *decryption* key only needs to be passed to the key store for 2 API calls. These are:

1. Return the unmatched portion of a specific bet. This is the remaining amount of the user's payment not yet matched when the event began.
2. Pay the user for their winning bet.

Because the database *decryption* key is passed from the betting exchange to the key store it highlights the need for secure communication between the two. This confirms the importance of the communication scheme we set out in section 6.3.4.

In order to minimize the potential vulnerabilities that can arise when the key store has access to the *database decryption key*, we minimize the amount of time that the key is held in memory. We do this by:

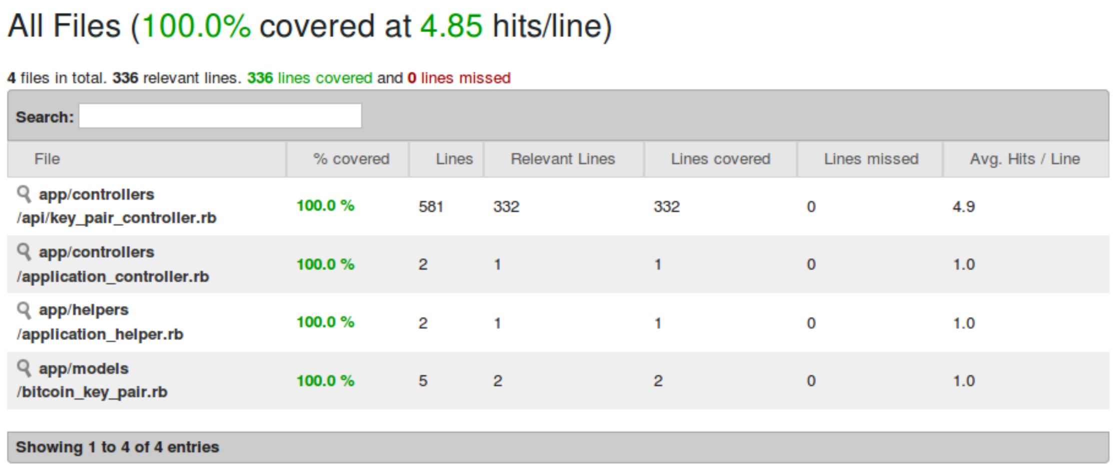
1. Only ever decrypting the *database key* as the very final step in our implementation. Remembering that we encrypt the parameters passed to the key store, it means we can keep the *database key* encrypted for as long as required. This means that we validate everything about the API call before ever decrypting the *database key*. This makes it slightly more difficult for an attacker to try to extract the key out of memory because they will first have to construct an API call that is valid and deemed appropriate by the key store.
2. The second step we take to minimize the amount of time the *database key* is held in memory is to use it as quickly as possible and then to discard of it immediately. This means as soon as the *database key* has been decrypted, we sign whatever transaction we need to, and then instantly remove the key from memory. We do this by first writing over the variable that stores the *database key* and then manually call the *Ruby Garbage Collector* to deallocate that memory. Figure 101 shows a code snippet of the API method that returns the unmatched portion of a users bet back to them. At this point in the code snippet, the API request has been fully checked and validated. The next step then is to decrypt the database key, sign the transaction, and then immediately overwrite the database key and manually call the *Ruby Garbage Collector*. This is seen in the snippet.

```ruby
# Return the unmatched portion of the bet back to the user
def sign_bet_return
  ...
    database_key = decrypt_api_parameter(params[:database_key])
    signed_transaction = sign_transaction(database_key, template_transaction)
    database_key = "WRITE OVER THE DATABASE KEY"
    GC.start # Manually Call the Garbage Collector
    render json: {"signed_transaction": signed_transaction}, status: 200
  ...
end
```

Figure 101: A code snippet of the method that returns the unmatched portion of a users bet back to them.

6.3.6 Testing

Given the critical nature of our key store and the API it provides, it was essential that we thoroughly tested and verified our implementation. Considering that a simple bug or mistake in our code could open up significant security vulnerabilities it was essential that we thoroughly locked down our implementation. As such we aimed to test all of the behaviours of our API and pushed for a very high code coverage on the API implementation. To do this we used the same test frameworks and tools we used to test our web application, *RSpec* and *simplecov*. Figure 102 shows a screen-shot of the output produced by *simplecov*. As can be seen we achieved 100% code coverage on our application. Given that the application did not contain many files or parts, the majority of our testing took place in the `key_pair_controller.rb` file, where the logic for our API lives. Although code coverage does not provide an absolute indicator of how well a file is tested, it does give us some idea about the areas that need improvement. As such we used *simplecov* to help guide us when writing tests for the most critical areas of our API.


All Files (100.0% covered at 4.85 hits/line)

4 files in total. 336 relevant lines. 336 lines covered and 0 lines missed

Search:

File	% covered	Lines	Relevant Lines	Lines covered	Lines missed	Avg. Hits / Line
app/controllers /api/key_pair_controller.rb	100.0 %	581	332	332	0	4.9
app/controllers /application_controller.rb	100.0 %	2	1	1	0	1.0
app/helpers /application_helper.rb	100.0 %	2	1	1	0	1.0
app/models /bitcoin_key_pair.rb	100.0 %	5	2	2	0	1.0

Showing 1 to 4 of 4 entries

Figure 102: A screen-shot of the code-coverage of our key store API.

Figure 103 shows a code snippet of a test for the API method that updates the total amount matched for a bet. The test in the snippet checks that setting the amount matched to a value lower than the amount already matched fails, returning a *400 bad request* error. As can be seen in the snippet, a bet is created that has received a payment of 10,000 satoshi. We set the amount matched for that bet to 5,000 satoshi. An API call is then made to update the amount matched to 4,950 satoshi. Because this new amount is less than the previous amount matched (which doesn't make sense because the amount matched should always be increasing) the API call fails.

```
...
  it "should not accept an invalid total amount matched: amount matched less than
    already matched" do
    encrypted_bet_address = encrypt_string(TEST_BET_ADDRESS)
    existing_bet = create_new_bet(TEST_BET_ADDRESS)
    set_bet_payment_received(existing_bet, "10000")
    set_bet_total_matched(existing_bet, "5000")
    new_total_matched = "4950"
    encrypted_new_total_matched = encrypt_string(new_total_matched)

    post '/api/update_matched', :format => 'json', :bet_address =>
    encrypted_bet_address, :total_matched => encrypted_total_matched
    expect(response.status).to eq(400)
    expect(json["error"]).to eq(TEST_INVALID_ARG_STRING)
  end
...
```

Figure 103: A code snippet of an API test that tests a call to update the amount matched for a specific bet. This test checks to see that setting the amount matched to an amount less than the current amount matched fails.

7. Evaluation

The *Bitcoin Betting Exchange* is the first publicly available betting exchange that uses the *Bitcoin Blockchain* as a way to announce *bet* and *outcome* proofs. It is the only betting exchange currently available online that provides complete transparency to its users, allowing them to use signed proofs and transactions as a way to verify the honest operation of the exchange. In addition, the *Bitcoin Betting Exchange* is the only betting exchange to provide complete user anonymity, requiring no personal information and no user sign-up.

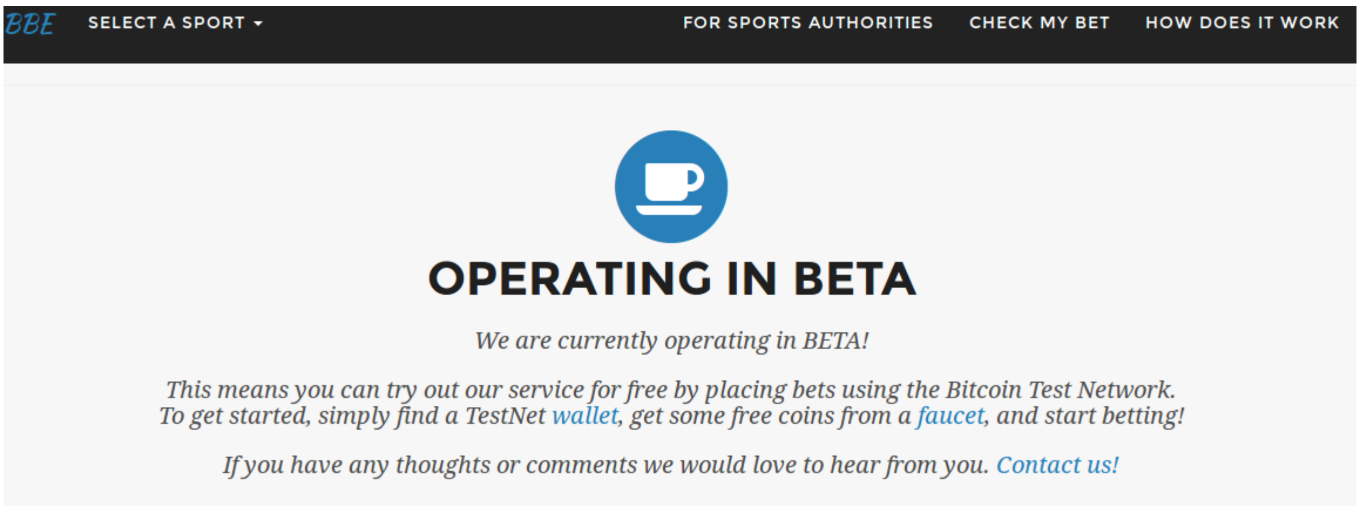
Furthermore, our *Smart Private Key Store* is the first *Bitcoin* wallet to use proofs embedded into the *Blockchain* as a way to determine whether or not a payment should be generated. This key store provides several interesting and attractive properties that give it improved security over generic *hot wallets*.

In order to effectively evaluate our proposed solution, there are two important areas that we must look at. The first is *user experience* and the second is *security*.

7.1 User Experience Evaluation

7.1.1 Online Feedback and Discussion

Given that our project lies in the intersection between *Bitcoin* and betting, we thought it best to solicit user feedback from users who are well versed in both of these areas. This is because users interested in both *Bitcoin* and betting are ultimately our target users and are the users we are most interested in getting feedback from. Once our solution was in an appropriate state, we publicly released the *Bitcoin Betting Exchange* online, under *BETA*. Figure 104 shows a screen-shot of the homepage of the *Bitcoin Betting Exchange* when it first opened for *BETA*.



OPERATING IN BETA

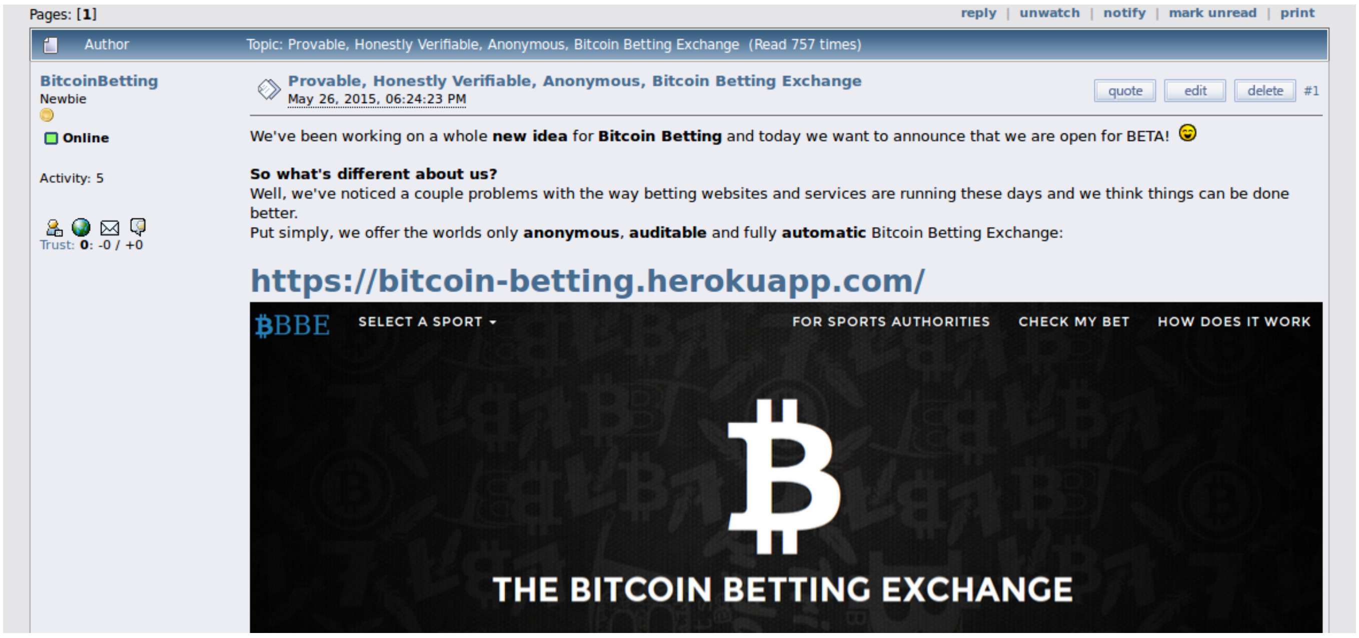
We are currently operating in BETA!

This means you can try out our service for free by placing bets using the Bitcoin Test Network. To get started, simply find a TestNet wallet, get some free coins from a faucet, and start betting!

If you have any thoughts or comments we would love to hear from you. Contact us!

Figure 104: A screen-shot of the *BETA* release of the *Bitcoin Betting Exchange*

To announce the release of the *Bitcoin Betting Exchange* to the *Bitcoin* community we created two posts online. The first was a post in the *Betting and Gambling* section of the *BitcoinTalk* [21] forum and the second was a post in the *Bitcoin subreddit* [98]. Figures 105 and 106 show screen-shots of these posts.


Pages: [1]

reply | unwatch | notify | mark unread | print

Author | Topic: Provable, Honestly Verifiable, Anonymous, Bitcoin Betting Exchange (Read 757 times)

BitcoinBetting
Newbie
Online
Activity: 5
Trust: 0: -0 / +0

Provable, Honestly Verifiable, Anonymous, Bitcoin Betting Exchange
May 26, 2015, 06:24:23 PM

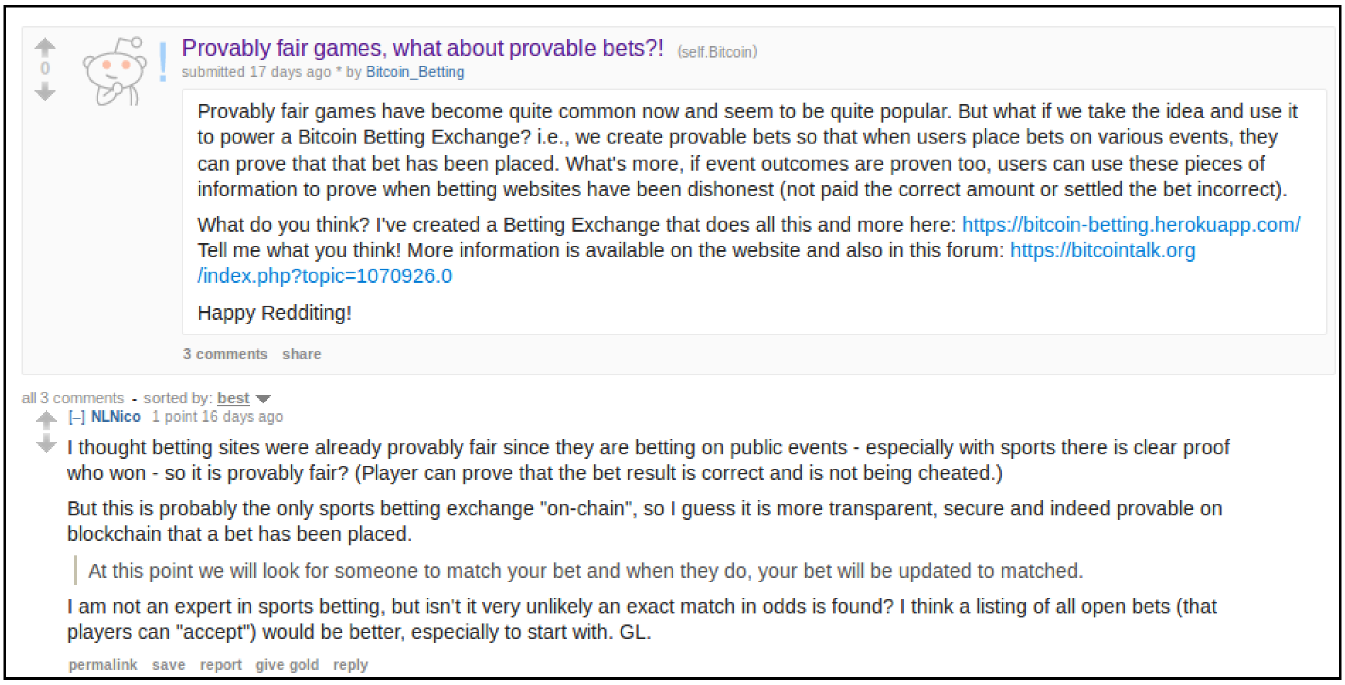
quote | edit | delete | #1

We've been working on a whole **new idea** for **Bitcoin Betting** and today we want to announce that we are open for BETA!

So what's different about us?
Well, we've noticed a couple problems with the way betting websites and services are running these days and we think things can be done better.
Put simply, we offer the worlds only **anonymous**, **auditable** and fully **automatic** Bitcoin Betting Exchange:

https://bitcoin-betting.herokuapp.com/

Figure 105: A screen-shot of the *BitcoinTalk* post about the release of the *Bitcoin Betting Exchange*.
`https://bitcointalk.org/index.php?topic=1070926.0`


Provably fair games, what about provable bets?! (self.Bitcoin)
submitted 17 days ago * by Bitcoin_Betting

Provably fair games have become quite common now and seem to be quite popular. But what if we take the idea and use it to power a Bitcoin Betting Exchange? i.e., we create provable bets so that when users place bets on various events, they can prove that that bet has been placed. What's more, if event outcomes are proven too, users can use these pieces of information to prove when betting websites have been dishonest (not paid the correct amount or settled the bet incorrect).

What do you think? I've created a Betting Exchange that does all this and more here: https://bitcoin-betting.herokuapp.com/
Tell me what you think! More information is available on the website and also in this forum: https://bitcointalk.org/index.php?topic=1070926.0

Happy Reddditing!

3 comments share

all 3 comments - sorted by: best

[–] NLNico 1 point 16 days ago

I thought betting sites were already provably fair since they are betting on public events - especially with sports there is clear proof who won - so it is provably fair? (Player can prove that the bet result is correct and is not being cheated.)

But this is probably the only sports betting exchange "on-chain", so I guess it is more transparent, secure and indeed provable on blockchain that a bet has been placed.

> At this point we will look for someone to match your bet and when they do, your bet will be updated to matched.

I am not an expert in sports betting, but isn't it very unlikely an exact match in odds is found? I think a listing of all open bets (that players can "accept") would be better, especially to start with. GL.

permalink save report give gold reply

Figure 106: A screen-shot of the *Reddit* post about the release of the *Bitcoin Betting Exchange*.
`https://www.reddit.com/r/Bitcoin/comments/37d28s/provably_fair_games_what_about_provable_bets/`

The purpose of these posts was to advertise our solution to the right types of users, those specifically interested in *Bitcoin* and betting in general. This was so that they might visit our application, hold conversations about it, and provide us with much needed feedback. In order to monitor user interaction with our website, we used *Google Analytics* [55]. *Google Analytics* is a web analytics service that tracks and analyses website traffic. It allows you to gain insight into the way that users interact with your application. For example, you can track the number of page views per day and the average amount of time a user spends on your application. By adding *Google Analytics* to our website, it enabled us to gather quantitative information and feedback about the way that users interacted with our application.

Looking at the posts we published online, we received some very useful feedback. In total our posts were read over 800 times, with 11 comments left by users. 5 out of the 11 comments ($\sim$ 45%) suggested interest in using our application but were unimpressed with the number of sports currently on offer (e.g. at the time we only supported around 5 different sports). 6 out of the 11 comments ($\sim$ 54%) mentioned that they specifically liked the design and layout of our application. 3 out of the 11 comments ($\sim$ 27%) mentioned dislike for the domain name. Understandingly all of these comments were made on the *Gambling* forum of *BitcoinTalk* and so as you would expect with a forum devoted to *Bitcoin* betting, the majority of these users were primarily focussed on placing bets. As such, their motivations for reading the post were more than likely to bet and not so much to discuss the unique ideas we propose.

Responding to this feedback, we increased the number of sports we provided from 5 to around 20. Unfortunately however, given the time constraints of the project, we were unable to move our hosting away from *Heroku* or to purchase our own *domain name*. Migrating to a better platform and custom *domain name* is something that will only be done after this project is complete. Interestingly however, 3 out of the 11 comments ($\sim$ 27%) mentioned that they specifically liked the unique aspects that our solution provides, such as *anonymity* and *bet proofs*. On the *reddit* post for example, one user discussed with us the design of our *proof of bet* and *outcome* system and agreed that it provides an advantage over traditional betting exchanges. This tells us that the features that make our betting exchange unique did not go unnoticed.

During the short period under which our site operated in *BETA* (around 3 weeks in total) we received around 1,660 page views from over 520 different users. Interestingly the mean session duration of each user was around 30 seconds in total, with a *bounce rate* of around 70%. Note that a *bounce* means that a user only reads the first page of the website and then leaves. One speculation as to why the *bounce rate* was so high is that our solution was only released on the *Bitcoin Testnet*. For many users who wish to bet using real *Bitcoin*, our website is not useful to them, and so once they see this limitation on the homepage, they leave the site. Figure 107 shows a graph of the number of website session per day during the *BETA* release. On the y-axis is the number of sessions, and on the x-axis is the date. Each dot represents a different day. The first red circle on *May 22* highlights the date our posts were made public and the *Bitcoin Betting Exchange* released. As can be seen the number of sessions grew steadily over time, stabilizing and then dramatically dropping on *May 30* at the second red circle. This dip was due to downtime we experienced when updating the number of sports provided by the application.

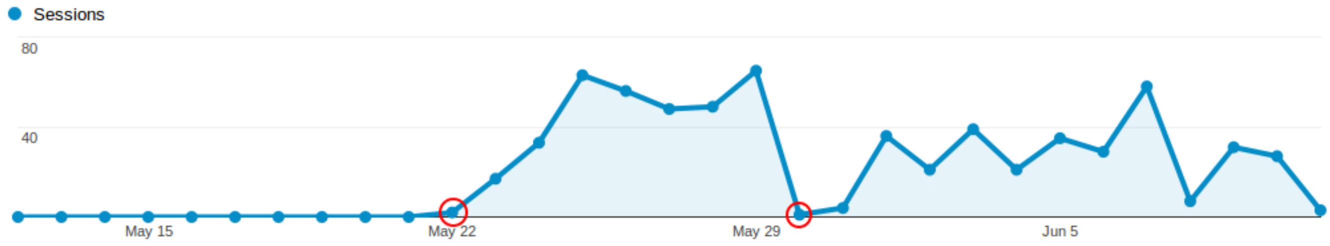
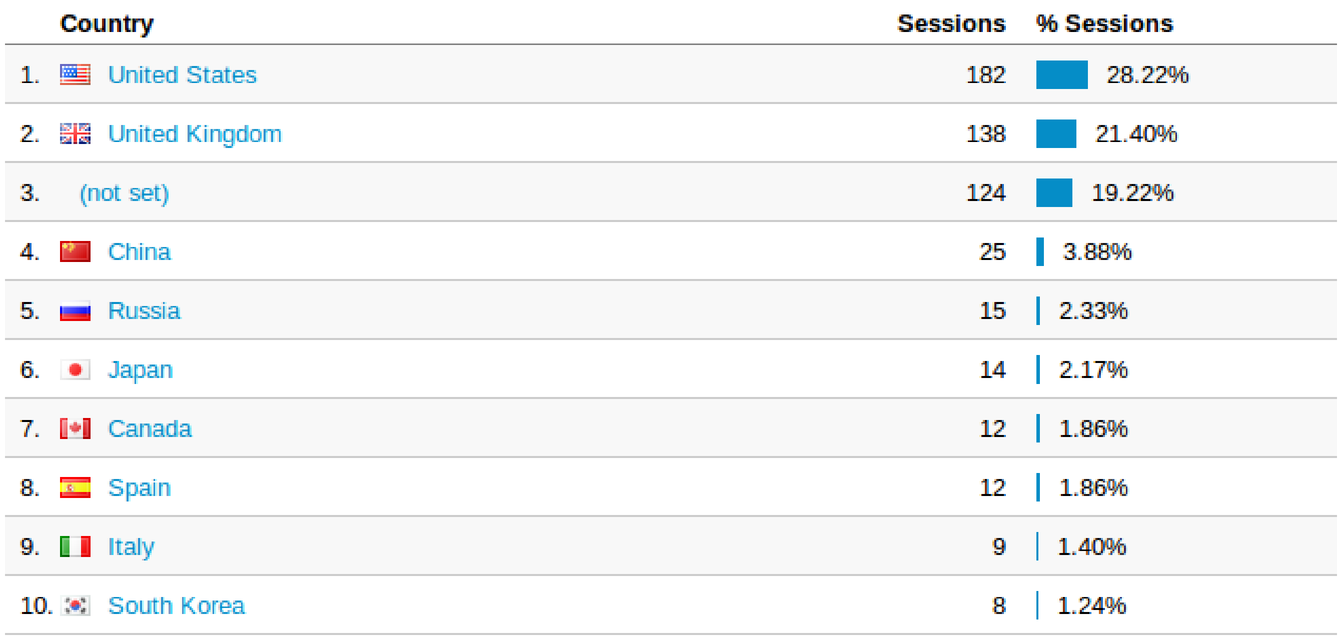


Figure 107: A graph of the number of website sessions per day during the *BETA* release.

One interesting observation we made during this period was about the locations of the users who visited the site. Figure 108 shows a table of the percentage of users coming from different locations around the world. As is immediately obvious, around 28% of all requests came from users in the *United States*. This is very interesting to note because *online sports betting* is illegal in the *United States*. This means that those users who visited our site were more than likely interested in pursuing illegal activity. Furthermore, 19 percent of user locations were hidden and 2% came from Russia where online betting is also illegal. Assuming the 19% percent of hidden locations were hidden for a reason, around 50% of all the traffic to our website was questionable. Only 21% of the users came from the UK.



	Country	Sessions	% Sessions
1.	United States	182	28.22%
2.	United Kingdom	138	21.40%
3.	(not set)	124	19.22%
4.	China	25	3.88%
5.	Russia	15	2.33%
6.	Japan	14	2.17%
7.	Canada	12	1.86%
8.	Spain	12	1.86%
9.	Italy	9	1.40%
10.	South Korea	8	1.24%

Figure 108: The percentage of users coming from different locations around the world.

During our public release we only had 9 bets placed by users who visited the site. Unfortunately, only around 3 of those bets were paid for using a *Testnet* wallet. Our suspicion as to why this number is so low ($\sim$ 33%) is because many users will not own a *Testnet* wallet, and because *Testnet* coins have no value, there is little motivation for them to get access to one.

Our takeaway points from this analysis are that firstly, our solution has potential. Having our posts read over 800 times, with 520 different users visiting our site, it tells us that our solution has generated interest from real users. Furthermore, the various comments and feedback we received on our posts in such a short period of time acknowledge and agree with this statement. Users also actively mentioned interest in the unique features that our solution provides. However, the biggest obstacle we face in getting users to engage with our site is the fact that it does not run on the production *Bitcoin* network. This was a decision we made at the beginning of our project to avoid any potential legal issues from arising. Although this does reduce the amount of interest currently generated in our application, we feel this decision was correct given that around 30% of all our user traffic came from locations where online sports betting is illegal.

In terms of performance and website speed, the average page load time during the *BETA* period was around 0.55 seconds, with the worst load time being around 5.2 seconds. Given that our solution was hosted online for free, we feel an average load time of anywhere around half a second is good. Furthermore, after looking into why the worst load time was so high, we noticed that this occurred when the server was under an abnormally heavy load. This was due to an excessive number of requests being sent to the server in a short period of time. This is discussed later in the *security* evaluation. Based on response times and speed we think that the performance of our application is acceptable.

In terms of usability and user experience it is clear that in order for our solution to compete with the already existing solutions we need to add support for as many sports as we can. Specifically, we should add *soccer*, as this was the number one requested sport to add to our exchange, requested in at least 3 out of the 11 comments ($\sim$ 27%). Likewise, if our solution was to be released fully online, we would need to migrate our application to better servers and purchase our own *domain name*.

7.1.2 User Experience Feedback

In addition to collecting feedback from users online, we also conducted an in-depth user study. This was done at the departmental *Project Fair* and gave us the opportunity to gather much more detailed and in-depth feedback about our application. To collect this feedback we performed 5 in-depth user interviews, each of around 15 minutes long. Each user interview had a short *pre-interview* and *post-interview* questionnaire with the remainder of the interview being an interactive session. In the interactive session users were asked to perform specific tasks using our application and were timed and monitored. The focus of this was to observe how the users interacted with our site without guidance or assistance. We encouraged all users to think out loud and recorded these sessions using a screen and microphone recorder.

The purpose of the *pre-interview* questionnaire was to assess each user's background and experience with *Bitcoin* and betting. Unsurprisingly, given the technical background of the students in the department, all 5 of our test subjects had heard of *Bitcoin* before. They all knew vaguely what it was and how it worked. Surprisingly however, none of them had ever used *Bitcoin* to send or receive a payment and none of them owned a *Bitcoin* account. In addition, for all 5 of our test subjects betting was an area they had never had any experience with before. None of our subjects had ever placed a bet online, or in person, and none of them had ever used a betting website or visited a sportsbook.

After conducting the user interview, we then gave users access to our website and asked them to perform a set of specific tasks. These tasks included creating a bet, paying for a bet, cancelling a bet, looking up a bet and verifying a bet. During this time we did not give assistance to the user (unless asked) and recorded their interactions with the website. The various insights we were able to gather from the interactive sessions were:

1. 4 out of the 5 users (80%) got stuck at the point just before placing a bet on an event. This was because they were confused with the *back* and *lay* terminology, having never used a betting exchange before. All 4 of those users proceeded to click on the *"?"* help button in an attempt to see if they could find out what *back* and *lay* meant. This help button did not answer their question however, and so we were forced to explain the terminology. Figure 109 shows the point at which these users got stuck.

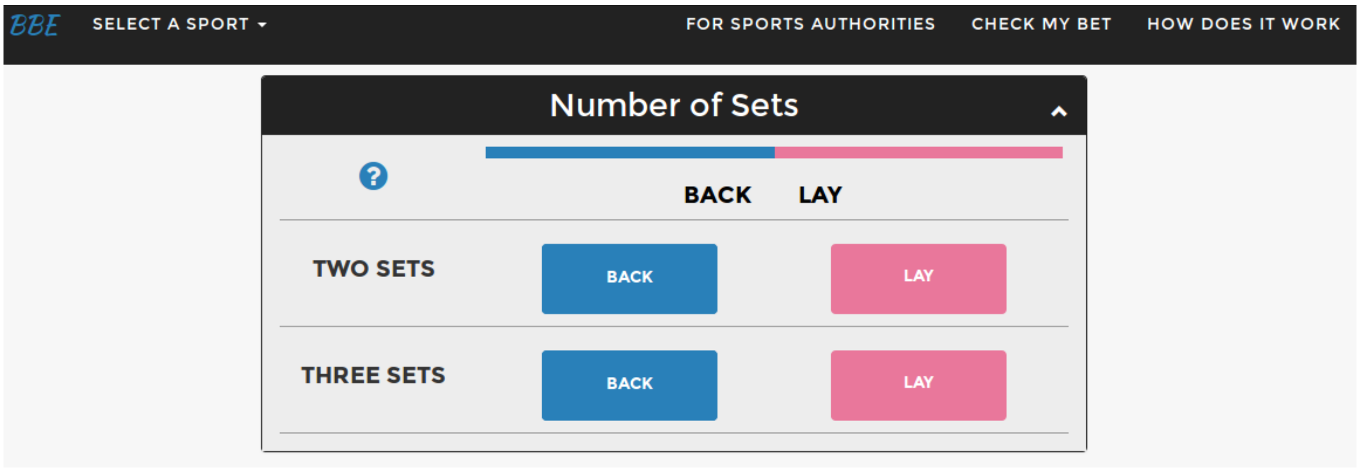
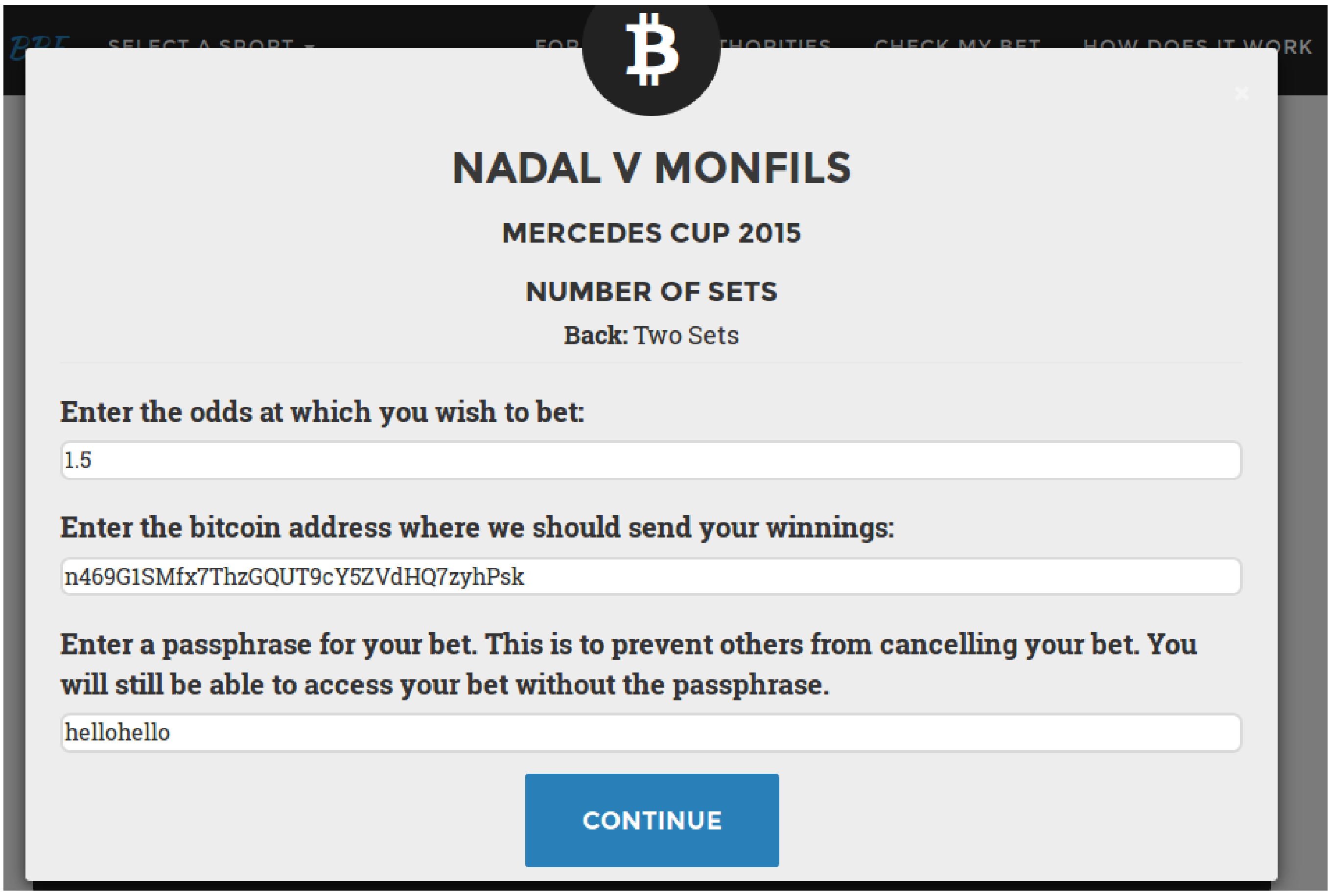


Figure 109: A screen-shot of the point at which 80% of our test subjects got stuck as they hadn't come across the terms *back* and *lay* before.

2. 3 out of the 5 users (60%) had to visit the *How it Works* page in order to clarify how the process of paying for a bet worked. After which, all 3 were able to pay for their bets without assistance from us.
3. 2 out of the 5 users (40%) struggled at certain times to navigate the website. This was because, after clicking a button, they weren't sure whether or not the website was loading or if the click hadn't registered and so ended up clicking multiple times.
4. 3 out of the 5 users (60%) commented that they thought the progress bar at the bottom of the page was a good idea, but 1 of those 3 struggled to see what stage of the bar we had progressed to. This was because they were uncertain about how the bar fills up.
5. 4 out of the 5 users (80%) commented that they liked the design of the website and felt that it was attractive.
6. 2 out of the 5 users (40%) commented that when entering a passphrase for their bet, they would like that passphrase to be covered up, or starred out. Figure 110 shows an uncovered passphrase.



NADAL V MONFILS

MERCEDES CUP 2015

NUMBER OF SETS

Back: Two Sets

Enter the odds at which you wish to bet:

1.5

Enter the bitcoin address where we should send your winnings:

n469G1SMfx7ThzGQUT9cY5ZVdHQ7zyhPsk

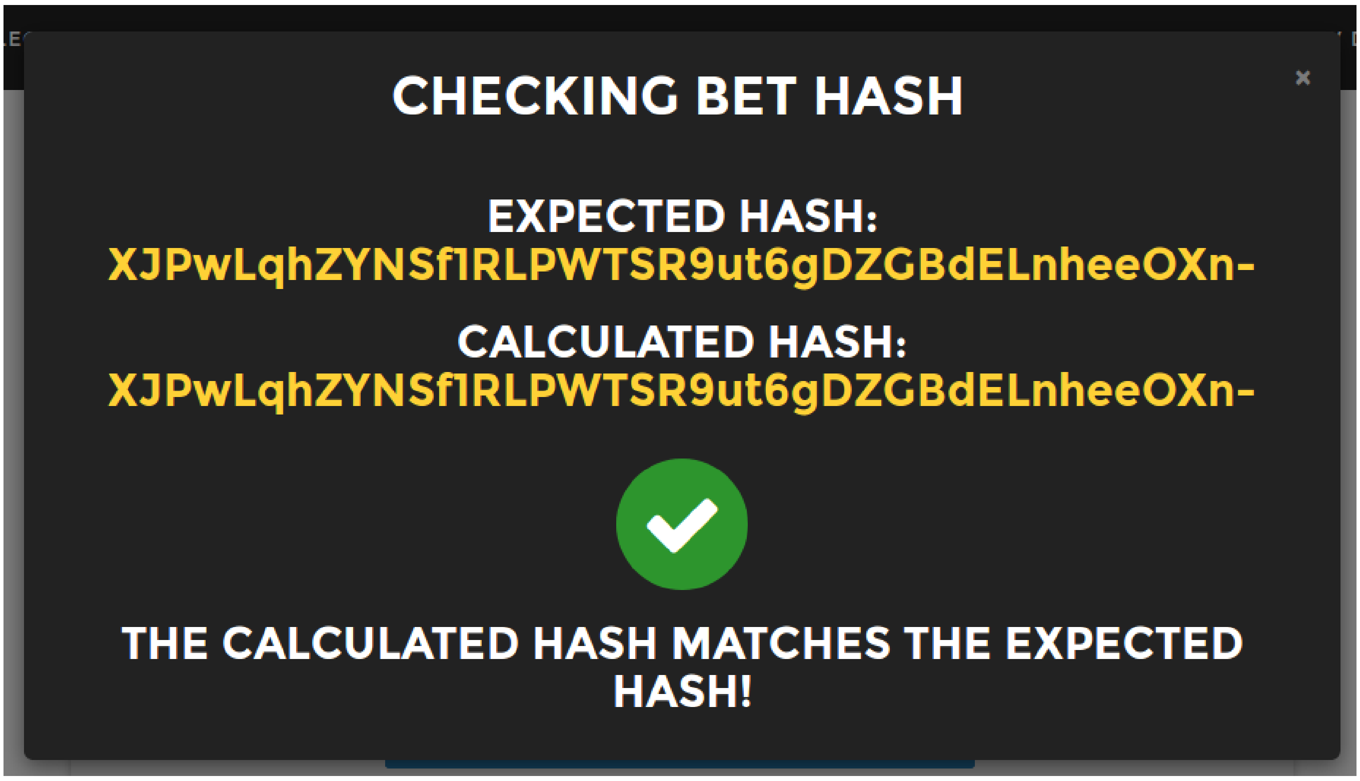
Enter a passphrase for your bet. This is to prevent others from cancelling your bet. You will still be able to access your bet without the passphrase.

hellohello

Figure 110: A screen-shot of an exposed passphrase when entered.

In the *post-interview* questionnaire we asked each user about the *proof of bet* and *proof of outcome* schemes that we use in our application. This was to assess how well they understood these proofs and whether or not our explanations were clear enough. In the discussions with our test subjects we found two important insights.

1. The first was that 80% of them (4 out of 5 of them) did not trust the in-browser validation that happens when a user clicks *"Validate My Bet Proof"*. Instead they all preferred to us the external website to validate this hash. This is because by not seeing exactly what the browser does, they have no way of knowing whether or not the check is actually being performed. Figure 111 shows a screen-shot of the validation result that the test subjects did not trust.


CHECKING BET HASH

EXPECTED HASH:
XJPwLqhZYNSf1RLPWTSR9ut6gDZGBdELnheeOXn-

CALCULATED HASH:
XJPwLqhZYNSf1RLPWTSR9ut6gDZGBdELnheeOXn-

THE CALCULATED HASH MATCHES THE EXPECTED HASH!

Figure 111: A screen-shot of the page that verifies that the bet information hashes to the correct hash.

2. The second insight was that 2 out of the 5 users found the ability to download the proof as a PDF reassuring. This was because even if they didn't fully understand the proof at the time, by downloading the PFD they will always have access to it, and so can always read more about how it works later on.

In response to this feedback we made several needed changes to our application. For example, we added a loading icon to the website so that users know exactly when the website is loading and can know whether or not to wait for it. We added a help section to the *back* and *lay* pages so that users who are unfamiliar with betting exchanges can find out about what these terms mean. We improved the progress bar by adding explicit text that lets the user know exactly what stage they are at. We also star out passphrases when they are entered into the website to prevent them from being read by someone else. Figure 112 shows a screen-shot of the loading icon added to the website. This icon is displayed in the top right hand corner.

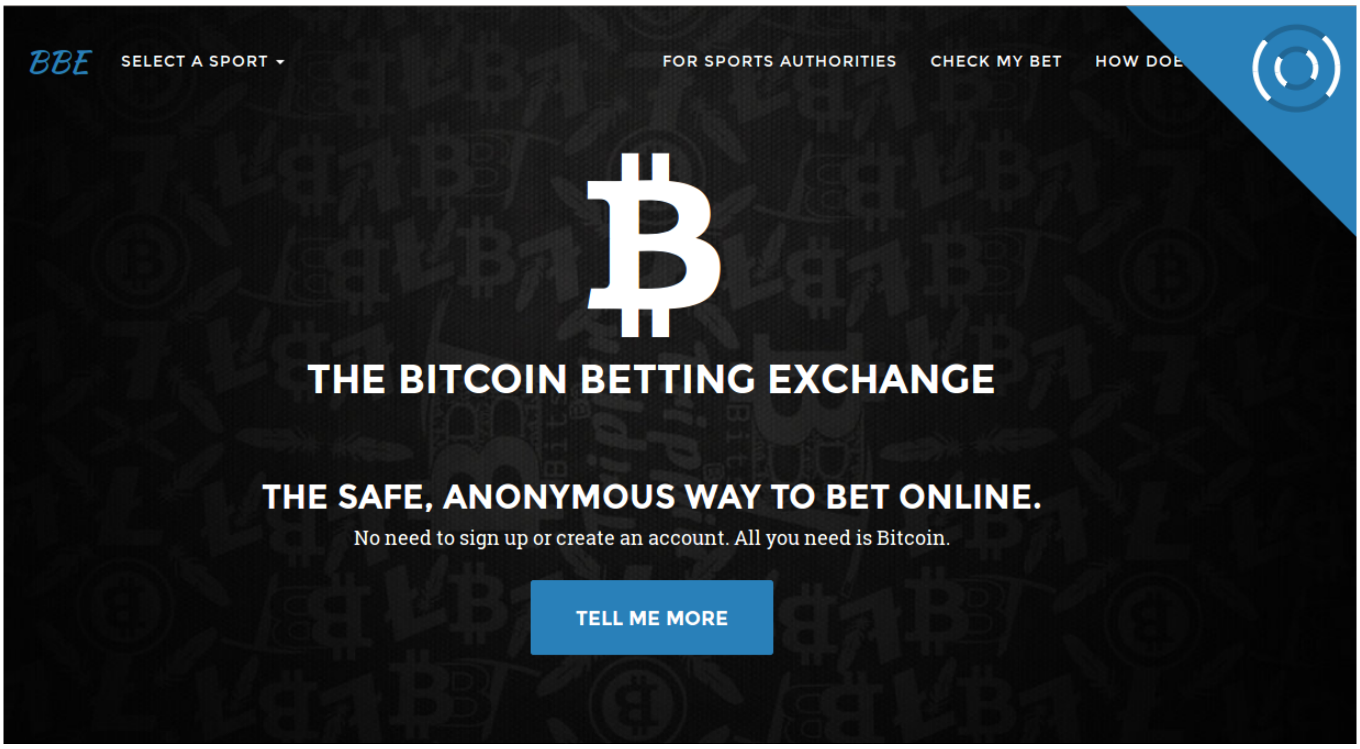

BBE
SELECT A SPORT
FOR SPORTS AUTHORITIES
CHECK MY BET
HOW DOE
THE BITCOIN BETTING EXCHANGE
THE SAFE, ANONYMOUS WAY TO BET ONLINE.
No need to sign up or create an account. All you need is Bitcoin.
TELL ME MORE

Figure 112: A screen-shot of the loading icon that is displayed when the web page is loading.

7.1.3 Professional Feedback

In addition to user feedback we also reached out to a company currently developing their own *Bitcoin* betting exchange. The founder of the company, *William Hanbury*, has been following our project closely from the beginning and so we were able to gather some useful insight from his side. Similarly to how we orchestrated our user tests, we showed *William* our working solution, allowed him to test it and had an in-depth discussion with him regarding our ideas. Some of the useful insights we found were:

1. From *William's* perspective, our *proof of bet* system and the way that we prove that a specific bet address corresponds to a specific bet is probably the most interesting part of our application. This was one issue *William* and his team had been trying to solve and the way in which we proposed to solved it incited interest.

2. In *William's* opinion, the *proof of outcome* system was not as interesting a problem to solve. This is because, from his experience, it is easy to find out who won an event, and so he had difficulty in seeing the benefit a *proof of outcome* provides. However, once we explained to him how our key stores uses these two *proofs* in combination, the value of the *proof of outcome* immediately became apparent because it enables our key store to protect its funds by acting as an adjudicator.

Overall, the feedback we received from *William* was very positive and as such, he expressed his interest in further exploring some of the ideas we had proposed together.

7.2 Security

Given that our project is security conscious, it is essential that we analyse it from a security perspective. This means studying the potential security vulnerabilities and issues that are present in our design and overall solution. It is worth noting here that our primary focus is not to find small implementation specific vulnerabilities (e.g. a bug in some component of *Rails*) but to instead analyse the larger security consequences of our design. Although we will no doubt find smaller implementation specific issues, these will need to be put into perspective regarding the overall solution.

7.2.1 Hacking Challenge

One of the first ways we wanted to evaluate our design was by issuing a *hacking challenge* online. This was a challenge inviting anyone interested in doing some *white hat* hacking to try to penetrate our website and key store. A challenge of this nature would not only test the robustness of our design, but also the implementation, and so it would allow us to gain insight into the vulnerabilities that exist. As such, we issued this challenge online as part of our post on the *BitcoinTalk* forum. The challenge invited anyone interested in hacking the site to contact us and we would provide them with an overview of the architecture as well as some insider information.

Unfortunately, we did not receive any formal replies to this challenge. We did however notice that a few days after it was published online, there were several attempts made at various different types of attack on our solution. The first of which was a *brute-force* attack. On the *27 May*, just 5 days after the post was published online, there was a 2 hour time window in which the server logs looked as follows (figure 113):

```
2015-05-27T15:13:16.847155+00:00 app[web.1]:    Rendered layouts/
      _bet_form_with_odd_input.html.erb (0.7ms)
2015-05-27T15:13:16.850310+00:00 app[web.1]:    Rendered layouts/
      _bet_form_with_odd_input.html.erb (0.7ms)
2015-05-27T15:13:16.853488+00:00 app[web.1]:    Rendered layouts/
      _bet_form_with_odd_input.html.erb (0.7ms)
2015-05-27T15:13:16.856608+00:00 app[web.1]:    Rendered layouts/
      _bet_form_with_odd_input.html.erb (0.7ms)
2015-05-27T15:13:16.859724+00:00 app[web.1]:    Rendered layouts/
      _bet_form_with_odd_input.html.erb (0.7ms)
2015-05-27T15:13:16.862865+00:00 app[web.1]:    Rendered layouts/
      _bet_form_with_odd_input.html.erb (0.7ms)
2015-05-27T15:13:16.866006+00:00 app[web.1]:    Rendered layouts/
      _bet_form_with_odd_input.html.erb (0.7ms)
2015-05-27T15:13:16.869194+00:00 app[web.1]:    Rendered layouts/
      _bet_form_with_odd_input.html.erb (0.7ms)
2015-05-27T15:13:16.876236+00:00 app[web.1]:    Rendered layouts/
      _bet_form_with_odd_input.html.erb (0.8ms)
```

Figure 113: A code snippet of our server logs under a brute-force attack.

As can be seen in the server logs, our server was continuously reloading the bet form every 0.7ms. During this time someone was trying to brute force the form that creates bets (this is the same form seen we have already seen where users create a bet and enter their passphrase). Given that when the attack was over no bets had been created, it indicated to us that whatever they tried had failed, and that the validation we put on this form was strong enough to prevent their attack.

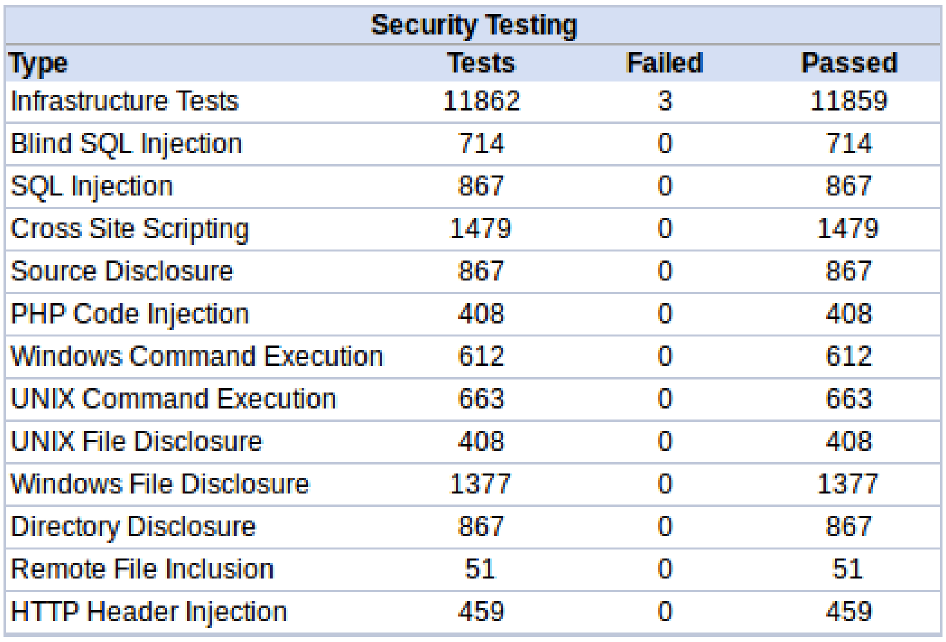
In addition, just two days later, on the *29 May*, a *denial-of-service* attack was made against our application. This was done by making an excessively large number of requests to the web server in a short period of time. These requests were more than our application could handle, and so it brought our betting exchange down for around 3 hours. It was during this period that we experienced the slowest response times. A *denial-of-service* attack however, does not indicate a serious vulnerability in our application as it is just a generic type of attack.

In order to prevent these types of attack from occurring in the future, we made use of the *rack-attack*[71] *Ruby Gem.* The *rack-attack* gem provides middleware for *Rails* applications that allows it to provide blocking and throttling of requests. Using the gem we limited the number of requests to our application to 5 requests per second, per *IP address.* Although this does not completely stop these types of attacks, it does slow them down enough to make it more difficult to effectively perform.

It is worth noting that several other *brute-force* attacks of the same nature were also made on various other forms, with no repercussions. Overall, the insight we gained from this was that the validation on our forms are appropriately secure to prevent SQL injections and invalid information being accepted.

7.2.2 Security Scanners

Another method by which we evaluated the security of our solution was to use dynamic and static security scanners. These are scanners that crawl through websites and source code looking for security vulnerabilities. The first scanner we tried on our solution was the *ScanMyServer.com*[108] web security scanner. We ran this tool on the *Bitcoin Betting Exchange*. This scanner searches for many types of security vulnerability, such as *SQL Injection, Code Injection* and *Cross-site scripting*. After running this scanner on our website, the result produced was pleasantly surprising. Out of a possible 20,000 tests run, only 3 failed. Figure 114 shows the summary produced. The 3 tests that failed were infrastructure tests of low severity. They were *Timestamp Retrieval*, *HTTP Inspection* and *HTTP Trace*. These three vulnerabilities could allow an attacker to estimate the uptime of the server, inspect certain details about the *HTTP* protocol used (such as the version), and reveal any proxies used. As such, all of the vulnerabilities were of low concern and categorized under *"intelligence gathering"*, meaning that no serious vulnerabilities were found. Running the web security scanner on our key store API produced no vulnerabilities.



Security Testing			
Type	**Tests**	**Failed**	**Passed**
Infrastructure Tests	11862	3	11859
Blind SQL Injection	714	0	714
SQL Injection	867	0	867
Cross Site Scripting	1479	0	1479
Source Disclosure	867	0	867
PHP Code Injection	408	0	408
Windows Command Execution	612	0	612
UNIX Command Execution	663	0	663
UNIX File Disclosure	408	0	408
Windows File Disclosure	1377	0	1377
Directory Disclosure	867	0	867
Remote File Inclusion	51	0	51
HTTP Header Injection	459	0	459

Figure 114: A screen-shot of the summary produced by *ScanMyServer.com* security vulnerability scanner.

Having tried a dynamic web security scanner, we decided next to use a static web security scanner. *Brakeman* [29] is a well-known and popular static vulnerability scanner built specifically for *Rails* applications. It analyses the source code of *Rails* applications and searches for common *Rails* vulnerabilities, such as calls to *eval* with unsanitized input, which would allow an attacker to execute code on the server. Running *Brakeman* on the source code for our key store API produced no errors, which was a good sign. Running *Brakeman* on the source code for the betting exchange, however, produced 2 errors. Both of these errors appeared to show a *cross-site scripting vulnerability* in one of our pages. This was the page that displayed *bet* and *outcome* proofs and

Brakeman complained that there was a *cross-site scripting vulnerability* in the link that allowed users to check their hashes on a different site. Figure 115 shows one of the two lines that flagged the vulnerability. In the code snippet we render a link (or button) that takes a user to the *online-convert.com* page where they can verify their bet hash. When we forward them to this page, we pass the string to hash as a query parameter. *Brakeman* flagged this as a vulnerability as it was unsure about whether or not the variable `string_to_hash` was user derived (e.g. set by a user) or set by us. If `string_to_hash` was user derived it would allow them to inject Javascript code into the page when the link is displayed. Fortunately however this was a false positive because the `string_to_hash` variable was in fact set by the server and could not be modified by a user. False positives of this nature are very common when using *Brakeman*.

```
 ...
<%= link_to "Check the hash through another website",
    "http://hash.online-convert.com/sha256-generator?string_to_hash=" +
  string_to_hash,
%>
 ...
```

Figure 115: A code snippet of our bet proof page that produced a false positive for *cross-site scripting* when analysed with *Brakeman*.

7.2.3 Security Audit

Given the limited success we had finding any serious vulnerabilities with the security scanners, we reached out to a *penetration tester and Internet Services developer*, *Charlie Hothersall-Thomas*. *Charlie* works at an Internet security and data mining company and has a keen interest in *Bitcoin*. As such, we felt that he might be interested to hear about our project and provide some insight relating to the overall security of our solution. Thankfully, he spent several hours of his time looking at our solution and trying to find security vulnerabilities in our implementation. *Charlie* was able to provide several useful insights as well as found two serious vulnerabilities in the betting exchange:

1. The first vulnerability he found was that he was able to view bets without needing access to the unique bet address of that bet. He did this by exploiting a bug in our implementation that allowed bets to be looked up by both *bet address* as well as *bet id* in the database. Naturally, given that bet ids typically start from 1 in the database and increase by 1 each time, he was able to get access to all of the bets by simply searching for bets using their database id. Although this vulnerability is very easily fixed, and has since been patched, if it was not noticed the implications would mean that an attacker could view and cancel all of the bets placed by other users. Even though this wouldn't allow the attacker to get access to the funds, it would still be an annoying problem for our users.

   Figure 116 shows a code snippet of the method that contained the vulnerability. When a user visits a bet directly though the url (e.g. `bitcoin-betting-exchange.com/bet/{BET_ADDRESS}`) the `show_bet` method below is called with the `BET_ADDRESS` value passed in as a parameter. Unfortunately, the call to `Bet.friendly.find(...)` supports both bet addresses as well as bet ids and so a user can simply visit a bet using the url (e.g. `bitcoin-betting-exchange.com/bet/1`). This was easily fixed by restricting the lookup to only work for bet addresses.

```
# The controller method called when a bet is shown
def show_bet
  @bet = Bet.friendly.find(params[:bet_addres])
  ...
end
```

Figure 116: A code snippet of the method that contained the vulnerability allowing users to access bets directly through the bet id.

2. The second vulnerability *Charlie* found relates to the implementation that hides (or stars out) users' passphrases as they type them in. Due to a bug in the Javascript, instead of sending the unstarred passphrase to be hashed and stored by the application, the stars are sent to the application. And so all passphrases end up being of the form *, **, ***, ****, etc. This allows an attacker to look up a bet by simply trying all lengths of strings made up of stars. Again, this bug was easily fixed. But if it had been unnoticed it would have allowed an attacker to access and cancel users bets.

3. One additional insight *Charlie* provided was that one of the weakest points of our *bet* and *outcome* proofs were the bootstrapping of keys. As mentioned, in order to prove that the private key of the *Bitcoin Betting Exchange* is a specific key (e.g. the key of the betting exchange is x), that key needs to be used to sign a document or statement, or it needs to be published online under a trusted key server. In addition, the keys of the sports authorities also need to be provided in a trustworthy manner as to prove that they are owned by the authorities themselves, and not by someone else. Due to the constraints of this project and the fact that it is currently operating on the *Bitcoin Testnet*, we have not requested key publication on a trusted key server. If however this project is released commercially and migrated to the *Bitcoin Mainnet*, these publications will be made.

The outcome of this evaluation was positive in general. *Charlie* noted that our design was well thought through and even though some vulnerabilities were found, none of them would allow an attacker to get access to our funds or our private keys. As such, this is a good indication that our system has been implemented to operate as designed.

7.2.4 Threat Analysis

In addition to analysing the security vulnerabilities of our implementation, it also necessary to analyse the potential vulnerabilities in our overall design. To do this, we need to think about how an attacker could abuse our system at a high-level and under what threat models they might be able to gain access to our private keys.

Creating Many Empty Bets

One way in which an attacker could abuse our system is by creating an excessively large number of bets that are never paid for. Because our *proof of bet* system creates a single *OP_RETURN* transaction for every new bet, creating this transaction costs us a single miner fee. Because this proof has to be embedded into the *Blockchain* before a user pays for their bet, an attacker could simply create many bets that they never pay for. As such, this would end up costing our application money. However, given that a single miner fee is currently around 10,000 satoshi and only worth £0.01, an attacker would have to create many new bets for the cost to be significant. Furthermore, in order to reduce the chance of this occurring we can limit the amount of unpaid bets for a single *IP address* per day. This will make it more difficult for an attacker to perform this type of attack. If however, this does not deter an attacker and the problem becomes more serious, we could implement a scheme where a user would have to deposit a small amount of money into each bet to cover the fee of the *bet proof*.

Collision Attacks

One type of attack that our application is susceptible to is a collision attack. As with all hashes, because the input space is larger than the output space, collisions can occur. In our application this means that if an attacker can find two separate pieces of bet information that hash to the same string, with one of those strings already embedded in the *Blockchain*, they can use the collision to prove that the betting exchange has been dishonest. For example, figure 117 illustrates how this might work. Imagine that Alice placed a bet on the *2015 Wimbledon final*, and when her bet was created the betting exchange hashed her bet information and embedded the hash into the *Blockchain* using an *OP_RETURN* transaction. Eve can use this *OP_RETURN* transaction as bet proof against the exchange. If Eve can construct some bet information that hashes to the same string as Alice's bet did, Eve can pretend that the hash embedded into the *Blockchain* for Alice's bet was actually for her bet. This means that if she sends a payment to the bet address in the fake bet information and the outcome of that fake bet was correct, she can use the *OP_RETURN* transaction and her payment to prove that the betting exchange did not pay her the winnings she deserves.

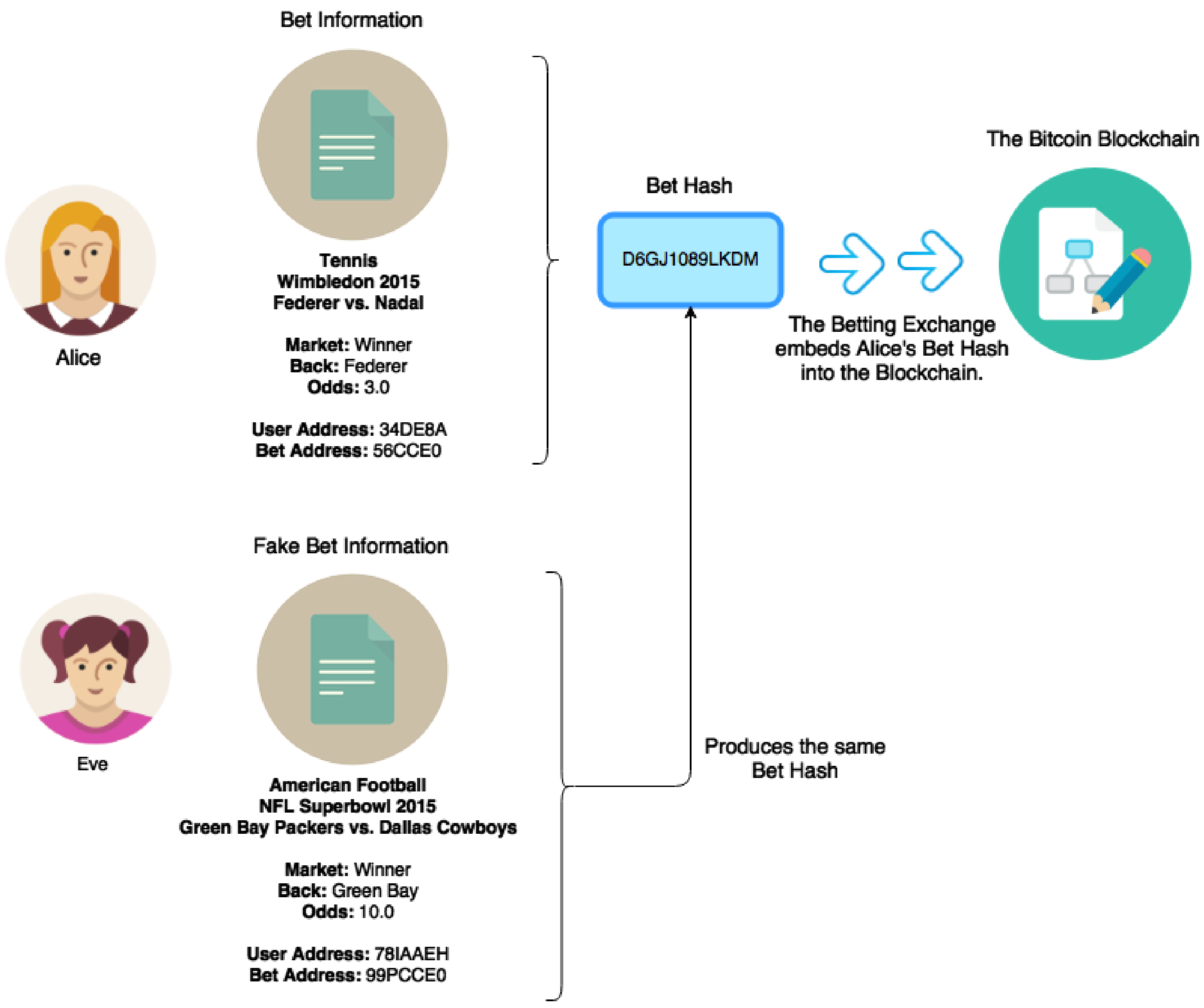


Figure 117: A diagram showing Eve perform a collision attack on the betting exchange.

Although this is possible, this type of attack is very difficult to perform from Eve's point of view. Firstly, the fake bet information that Eve constructs has to be in a valid format, for a real sports event, real market and real outcome selection. Also, the outcome in the fake bet needs to have occurred. Furthermore, the *Bitcoin* addresses in the fake bet information will both need to be valid addresses, with a payment existing that goes to the bet address. In addition, the payments she presents will need to be appropriately timestamped (e.g. she cannot present a payment made four years ago before the event was even announced). All of these constraints make performing this type of attack much more difficult. In order to push back against these attacks we must ensure that the hashing algorithm we use to hash bet information has not been broken. In addition, with the recent introduction of 80 character *OP_RETURN* transactions, increasing the size of the output space from 40 characters to 80 characters will make this type of attack even more difficult. Likewise, a similar sort of attack is possible for an *outcome* hash announced by a sports authority. But again, this is very difficult to perform.

Multiple Bet Payments

Another way in which an attacker could try to attack our system is by sending multiple bet payments to the same bet address. Given that our implementation only supports a single payment to a bet address, an attacker might try to send multiple payments to the address and claim wrong doing. For example, if an attacker places a small bet, pays for that bet and sees just before the event has finished that their bet will win. They might try to send another, much larger payment to the address and claim that they were paid the incorrect amount when the bet is settled. They would make this claim by presenting their bet proof, the larger payment to the bet address and the outcome proof. Although an attacker might think they could get away with this, they will be caught out. This is because every transaction in the *Blockchain* is timestamped and so using the timestamp of the second payment we could show that the payment was actually sent after the event began and so was not valid for matching. Likewise to prevent this being a problem for our users, any additional payments sent to a bet address are simple returned. This means that we can always show that subsequent payments made for a bet are returned, preventing attackers from trying to perform these types of attack.

Matched but the User Claims Cancelled

Another way in which a user might try to abuse our system is by claiming that they cancelled a bet when they in fact didn't. For example, imagine that a user places a bet, has that bet matched, but ends up losing the bet. The user might then claim that they cancelled their bet before it was even matched at all but the exchange refused to cancel it. Although a user could do this, and that this is a problem in any betting exchange, we can push back against it. Firstly, we can show that actually their bet was ultimately matched with someone else as it was used to pay a winning bet to someone else and so we didn't just keep the money. What's more we can also use the bet proofs to show that the bets that were matched together were in fact appropriate to match together and were matched in a timely fashion. Although this doesn't totally disprove their claim, if attacks of this type become a significant problem we can implement some form of *cancellation* proof, where the user who paid for the bet embeds a *cancellation* message into the *Blockchain* using the account used to pay for the bet. This will timestamp their *cancellation* and allow them to prove that we didn't cancel their bet. Likewise, it will also protect users from making false claims against us as they will have had to embed this *cancellation* message into the *Blockchain.*

Tracing Bet Matches

Another potential concern regarding our design is that it is possible for users to see who their bets have been matched with. When a bet is settled and the winner is paid, the winner is paid using the payments of the losing bets and so it is possible for users to track their payments. This means that users can see which bet addresses their bets have been matched with and which *Bitcoin* accounts have paid into those addresses. Although this isn't a vulnerability or an attack in itself, it does allow users to see who they have been matched with and so they can tell which *Bitcoin* addresses have been used to place a specific bet on the exchange. However, this is the only thing they can infer and they cannot tell who owns those accounts. One way in which to prevent this is to have every user pay into a single *Bitcoin* address owned by the exchange. That way, with enough bets going in and out, it is difficult to track which payments have been matched together. Doing this however, means that you can no longer prove that a payment into a specific address is a payment for a specific bet, and so the *proof of bet* would no longer work.

Double-Spend Attack

One possible attack to take into consideration is that of a *double-spend* attack. As previously mentioned, a *double-spend* attack is where an attacker is able to spend the same input twice, in order to steal an item or use a service. This can be done by spending the input again before it has been mined by the network, or by creating a *Bitcoin* fork and constructing a new *Blockchain* longer than the original. As discussed, when *SatoshiDice* originally opened they were very susceptible to *double-spend* attacks. This was because they did not wait for the user's payment to be accepted by the network before releasing the outcome of the user's bet. This allowed users to quickly create a *double-spend* transaction with a high mining fee that sent the same input back to themselves. This was in the hope that the network might mine the second transaction before the first and so they would not lose their money.

Due to the way in which our solution works, it is possible for an attacker to execute a *double-spend* attack on our exchange. The likelihood of this occurring depends on how quickly an attacker can determine whether or not their bet lost. For example, imagine that an attacker places a bet right before the beginning of a horse race, and that their bet is immediately matched. If the horse being bet on falls and is disqualified when the event beings, the attacker knows immediately that their bet will have lost. As such, they can create a *double-spend* transaction with a high mining fee in the hope that the network will mine it before the original payment. Alternatively, if the attacker has a lot of computing power, they might create a fork of the *Blockchain* longer than the original and mine the new transaction instead of the old one.

The probability of an attacker performing this type of attack depends on the length of time between when the bet is placed and when the attacker knows the outcome of the bet. The longer the period between these two points, the less chance there is for an attacker to perform a *double-spend* attack. This is because as the time between the two points increases, so does the computing power required to re-solve the blocks in the *Blockchain*. As such, to defend against this attack we can propose stipulations on the times up to which we accept bets for certain events. These stipulations will depend on the length of the sporting event as well as the amount being bet. So, for example, really large bets placed on quick events (such as horse races) might have a stipulation saying that bets are only accepted up to an hour before the event begins. In contrast, very small bets placed on longer events (such as football matches) might have a stipulation saying that bets are accepted right up until the event begins. This time constraint can be therefore be a function of the event length and bet amount.

Replay and Man-in-the-Middle Attacks

Another security concern of ours is *replay attacks*. A *replay attack* is when an attacker re-sends or delays a valid message sent by a client to a server. This is in the hope that they might exploit some vulnerability present in the way that the systems are implemented. Our solution guards against *replay attacks* in several different ways. Firstly, our implementation requires requests to be authenticated when sent from the client to the server and vice versa. These authentication signatures expire after 15 minutes and so an attacker would have to replay a message within that period. Furthermore, because our implementation requires a response from the server when a client makes a request, if an attacker were to prevent a message from being received by the server, the client would simply retry again. Secondly, because communication between the two parties uses *HTTPS* over *SSL*, the *MAC* used in each request includes a sequence number and so the server would not accept a request it has already seen before. Similarly, because communication between the client and server occurs using *HTTPS* over *SSL* we can avoid *man-in-the-middle* attacks. This can be done by always ensuring that we communicate over *HTTPS* (e.g. that we are not redirected to an *HTTP* connection) and that we always validate the certificate presented (e.g. don't accept incorrect certificates or trust an untrustworthy *certificate authority*).

Physical Access to the Hardware Device

Perhaps one of the most straight forward ways for an attacker to gain access to our keys is to try to attain physical access to the hardware device. Given that the device will be plugged in to the server running the betting exchange, for an attacker to steal it, they will need access to the location where the server is running. Because this area is a high security area with controlled and monitored access, it will be hard for an attacker to steal the device unnoticed. Furthermore, because the private keys are encrypted when written to the device, the attacker will also need access to the decryption key. As such, this will require the attacker to either steal the key from the betting exchange or wait until the key is passed to the device and decrypted in memory. Requiring both the device and the decryption key make it difficult for an attacker to read the keys off of the device.

Private Keys of the Sports Authorities

Another critical aspect of our design is the security of the private keys owned by the sports authorities. As previously discussed in section 4.2.5, attention needs to be given to the way in which the sports authorities manage and store their keys. If an attacker can get access to these keys, they can announce incorrect outcomes into the *Blockchain*. This would allow them to incorrectly settle bets in their favour. However, under the recommendations we made for the secure storage and management of these keys, an attacker would need physical access to the *Bitcoin Trezor* device used to hold them. Furthermore, they would also need access to the passwords set on the device and so this type of attack would be difficult to perform.

Seed Theft

The private key *seed* used to generate the private keys is perhaps the most critical link in our entire system. Using a *seed* to generate new *Bitcoin* addresses allows us to keep a back up of our private keys without ever having to copy them off the private key store. However, it does mean that whoever has access to the *seed* on the device, or the back up of the *seed*, can derive the private keys and so would be able to spend all of the money in our *Bitcoin* accounts. As suggested, to make it as difficult as possible for an attacker to get access to the *seed*, we recommend storing the back up safely. This can be done using a *paper wallet*, in a vault, under several layers of encryption. The decryption keys for that *seed* could then be fragmented and distributed amongst several trusted parties. Likewise, to prevent weak *seed* attacks, the chosen *seed* needs to be strong and generated appropriately. Therefore for an attacker to gain access to the *seed*, they would either need direct access to the hardware device or they would somehow have to get access to the physical back up of the *seed*. Given that both of these will be stored in high security areas, it makes this type of attack incredibly difficult.

Control of the Web Server

Under a threat model where an attacker can execute all 5 API calls from the betting exchange and can communicate with the web application's database, it is interesting to note that there are not many attacks that can be performed:

1. As already discussed, an attacker could keep making API calls to generate a new bet address and bet proof. The worst this would do however is cost the exchange money due to the

number of *OP_RETURNS* generated. The attacker wouldn't be able to get access to any funds.

2. An attacker could cancel a users bet by making an API call to return the unmatched portion of the bet back to the user. Again however, this wouldn't allow the attacker to gain access to any of the funds because they cannot change the address to return the amount to. Instead the amount would be returned to the address originally given when the bet was created.

3. An attacker could increase the amount matched for a bet regardless of whether or not that amount was actually matched. Although this would be annoying, it would not allow the attacker access to any funds. Instead it could be fixed manually by waiting for the bet to be settled and then manually signing a transaction returning the remainder of the unmatched portion back to the user.

4. In fact, the worst an attacker could do is equivalent to a double spend attack. This attack is somewhat intricate and relies on the fact that the private key store does not have an active connection to the *Bitcoin* network, and so it doesn't know which transactions have been mined and which haven't. An attacker with access to both the website and the website's database could create a bet on an event. This would be through the API and the attacker would broadcast the bet proof into the network as per normal. Next the attacker would sign and create a transaction paying into the bet. Note that he wouldn't broadcast this transaction to the network but would instead just send it to the key store, telling the key store that a payment has been received. Then the attacker would wait for his bet to be matched by the application and this would would occur as per normal.

   When the time comes for the bet to be settled the attacker is in a special position. This is because if his bet lost, he would simply discard the bet payment he signed, making it so that it never existed. When the betting exchange tries to broadcast the winning transaction to the network, paying the other party, the network would reject it as it never saw the original bet payment made by the attacker. If however the attacker's bet won, he would broadcast his original payment to the network and then his winnings would be sent to him as per normal. This allows the attacker to protect himself from losing bets. However, given that this requires the attacker to be able to execute all the API calls from the website and to communicate with the website's database, we feel that the vulnerability is not that bad. This is because given that the attacker has access to such a large portion of our system, if the worst they could do is prevent themselves from losing a bet (but not get access to anyone else's funds) this trade-off is much better than a normal *hot wallet*.

   To protect ourselves from this attack, we could add a requirement to the API that says in order to set the amount paid into a bet, you have to present the signed transaction sent to the bet address as well as a signed statement that says how many confirmations that transactions has in the network. This statement could come from an external server that the betting exchange communicates with in order to fetch the current number of transactions. The private key store could be bootstrapped to trust the external server and that would prevent an attacker from not broadcasting their payment to the network. In order to perform the same attack, an attacker would have to somehow get a signed statement from the external server that says his transaction has a valid number of confirmations in the network, which makes the attack more difficult.

7.3 Strengths & Limitations

In this thesis we present the *Bitcoin Betting Exchange*, the first publicly auditable, anonymous and fully automatic betting exchange of its kind.

The main strengths of our work are:

- Our betting exchange is fully transparent and provably honest. It is the only betting exchange that protects its users and itself by publicly announcing *bet* and *outcome proofs*. This means that dishonest behaviour by any of the parties can be identified and shown.
- At the same time as being fully transparent, the betting exchange protects the anonymity of its users. It requires no personal information and no user sign up. What's more, even in the case of disputes where a bettor argues against the exchange, the identities of the parties do not need to be known.
- Interaction with our betting exchange is simple. It doesn't require the user to create special *Bitcoin* transactions or to communicate with several *oracles* in order to spend its money. Instead the user pays for their bet using their *Bitcoin* wallet and their winnings are automatically paid to them. No deposit is required and the amount of time we hold their funds is minimized.
- By not requiring the services of a third party *escrow* and by operating solely on the *Bitcoin* network we can offer competitive odds that are uncapped and unconstrained, with immediate resolution and automatic payout.
- Under a threat model where an attacker has access to a significant portion of our system, such as the betting exchange and the database, our private key store provides improved security over generic hot wallets. Furthermore, it operates automatically and requires no manual topping up or human intervention.

Although the work presented in this thesis has many strengths, it is not without its limitations:

- It is still possible for the betting exchange to steal a user's payment or refuse to settle a bet. Should it wish, nothing physically prevents it from acting dishonestly because during the bet process it has full access to the user's payment. However, as soon as it acts inappropriately, the user will be able to prove it and the reputation of the betting exchange will be destroyed. This means that no users will ever trust the site again.
- The speed at which payments can be sent to and received by the exchange are ultimately limited by the speed of the network. This means that depending on how many confirmations you require for a transaction, we have to wait for the network to mine these transactions and create new blocks in the *Blockchain*. Unfortunately, we are bound by this speed and at present there is no way to quicken this process.
- Event and market outcomes have to be manually embedded into the *Blockchain* by a sports authority. This means that there is still some manual intervention required in order to settle a bet. However, you could argue that regardless of the solution, sports authorities still have to do this anyway and so the same problem ultimately exists with any other betting website or *oracle* service. At some point or other the outcome of an event and market will still need to be published manually whether it is on website, API or by hand.
- Our solution relies heavily on the *Bitcoin* network and generates many *Bitcoin* transactions for a single bet. This means that for every proof we create, payment we accept, and settlement we pay, we place a load on the network to mine those transactions. One of the main concerns for the *Bitcoin* community at present is how the network will respond to growth in the future. Understandably, if many applications adopt proof mechanisms such as the ones we've proposed, the load placed on the network will be significant and there will pressure on the *Bitcoin* development community to find a way to support it.

8. Conclusion

8.1 Lessons Learnt

Throughout the duration of this project we have faced many interesting challenges. Facing these issues first hand has allowed us to gain insight into the way that *Bitcoin* works and the problems it is currently facing. Notably, we have seen just how powerful the idea of a *Smart Contract* is, but we have also learnt that *Smart Contracts* currently face strict limitations. The idea of building a provably secure contract directly into the primitives of *Bitcoin* are exciting, but until a way is found to effectively communicate with the outside world, these contracts are constrained and impractical for the purposes of a betting exchange.

In addition, we have witnessed the benefits that *m out of n* transactions and *Bitcoin oracles* provide, but also learn that until a stable ecosystem exists in which these *oracles* can be questioned, the impracticalities of *m out of n* transactions for automatic settlement are extremely high. Furthermore, we have also identified that these types of transactions severely impact the way in which a betting exchange can match bets. This is because the *n* individuals in the *m out of n* transactions need to be known before a payment can be sent and so the betting exchange cannot dynamically match bets without manual user intervention.

Surprisingly to us, we have also seen just how attractive *Bitcoin* is for criminal activity. With around 30% of all user requests coming from locations in which *sports betting* online is illegal, it has opened our eyes to the legal and social ramifications of operating an anonymous *Bitcoin* betting exchange online.

We have also been able to see the benefits that specialized *Bitcoin* wallets and hardware devices provide over generic *hot wallets* and *multi-tiered* approaches. Using proofs embedded into the *Blockchain* we can build a device that is resilient to many types of attacks. Regardless of the number of security layers present in a system there is always a weakest link.

Finally, we have seen first hand just how quickly *Bitcoin* and its ecosystem are evolving. Even throughout the duration of this project there have been several significant changes made to the way in which *Bitcoin* works. Even now, *Bitcoin* is undergoing many changes in order to support the demand that will be placed on it in the future.

8.2 Future Work

Given additional time and resources there are several improvements and extensions that can be made to this project:

- The first improvement we propose relates to the market liquidity of the exchange. As originally mentioned, one of the reasons we chose to use *Betfair* as our source of sport and market data is because it allows us to integrate the *Betfair* betting exchange with our own. This means we will be able to match bets across *Bitcoin* and fiat currency, similarly to how *DirectBet* currently do. This improvement would increase the volume of bets on our site and therefore make it more attractive to potential users. Furthermore, assuming that *Betfair* agree with the settlements published by the sports authorities (which they should, as the terms and conditions are exactly the same), this will not affect the proof of *bet* and *outcome* system.
- The second improvement we can propose for our solution is to design and implement an API for the betting exchange. This API will allow developers and bettors to build automated services and programs around our ecosystem. For example, using the API they could fetch the events and markets currently available, see the bets that are eligible for matching and place their own bets. This would make the exchange more accessible to the outside world and help to increase the number of bets available.

- Another improvement we would like to make to the project is to use *Chain notifications* [33]. At present our application polls various *Bitcoin* APIs to see the latest transactions, events and outcomes in the network. Instead we could make use of the notification service that *Chain.com* provide. This service would notify our application whenever an action takes place on the network that we are interested in, such as a payment to a specific bet address or an *outcome* announcement. This will avoid unnecessary polling and make our application more responsive as it will be notified immediately when the action occurs. It is worth noting that the reason we decided not to use this service originally is because the notification system is currently only available in *BETA* and provides no guarantees regarding uptime.

- Another improvement we propose is to update the length of the *bet* and *outcome proof* hashes. As mentioned, two thirds of the way through our project the *Bitcoin* developers announced a change to the number of characters supported by the *OP_RETURN* [39] transaction. In this change they increased the size of the *OP_RETURN* string from 40 characters to 80 characters to allow more data to be stored in the *Blockchain*. Due to the time constraints of our project our implementation still operates under the 40 character limit and as such, one way to improve our system would be to update the number of characters embedded into the *Blockchain*. As discussed in the security section, this will make our solution more robust to collision attacks.

- In order to make the *Bitcoin Betting Exchange* available online and run using the *Bitcoin Mainnet* there are several improvements and administrative tasks that need to be performed. Firstly, we will need to purchase our own *domain names*, *SSL certificates* and migrate to paid hosting with increased capacity. Secondly, we will need to publish our *Bitcoin* keys online using a *trusted key server* or by presenting a signed certificate to the users along with our *SSL certificate*. Thirdly, we will need to enforce location specific blocking and require users to provide their age and personal information in order to bet. This is to prevent the exchange from encouraging illegal activity. And finally, to see our system in its full form, a prototype of the *Smart Private Key Store* will need to be built in hardware .

A. Bitcoin Betting Exchange Entity Relationship Diagram

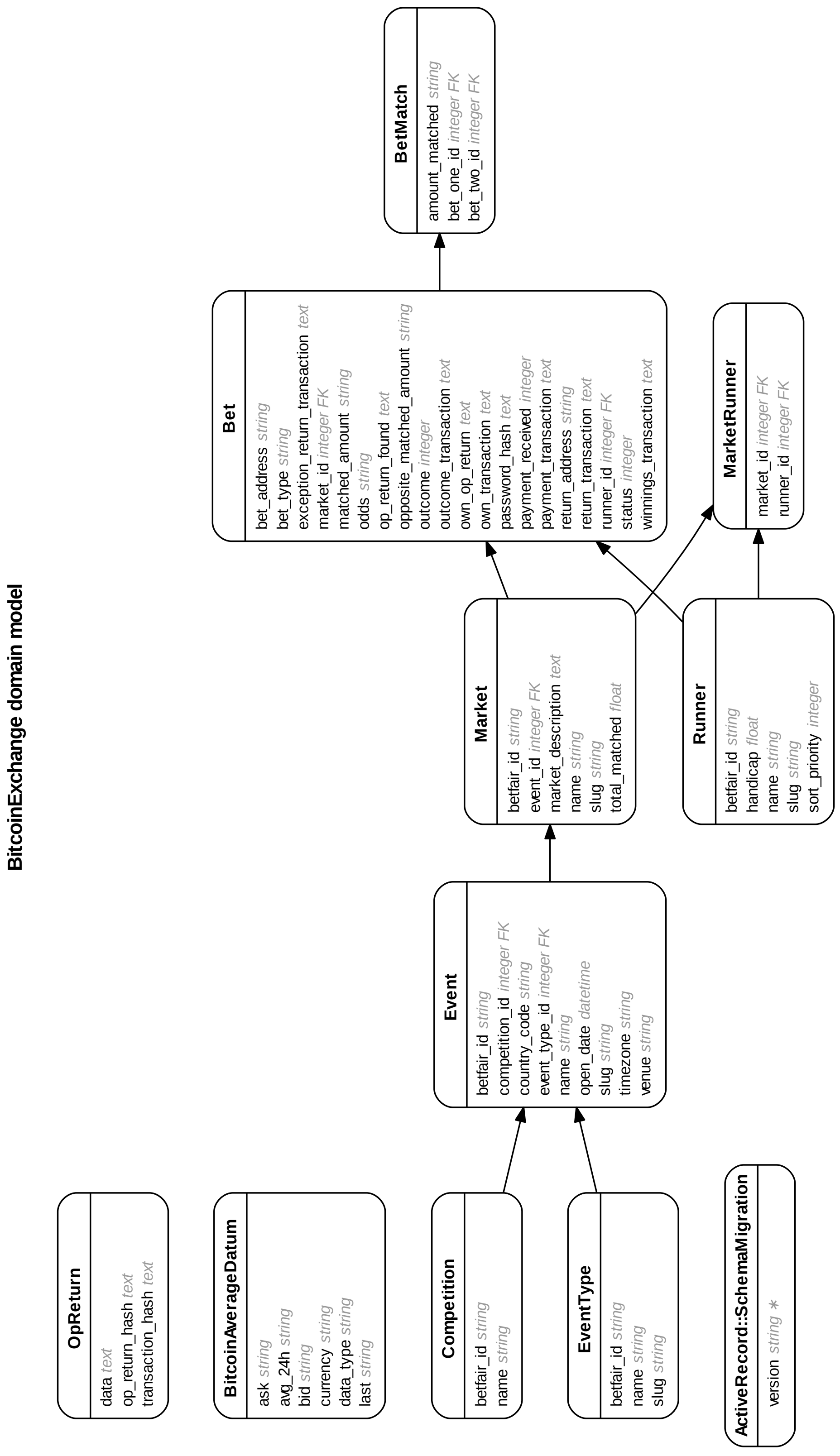


Figure 118: Entity Relationship Diagram for the Bitcoin Betting Exchange.

References

[1] 87th United States Congress. *Interstate Wire Act of 1961.* 1961. http://www.gpo.gov/fdsys/pkg/STATUTE-75/pdf/STATUTE-75-Pg491.pdf, Last accessed: 24 January 2015.

[2] Rediff India Abroad. *Nadal in final after Djokovic retires.* http://www.rediff.com/sports/2007/jul/07nadal.htm. Last accessed: 24 January 2015.

[3] Igor Alexandrov. *Ruby Gem: WisePdf.* https://github.com/igor-alexandrov/wisepdf. Last accessed: 9 June 2015.

[4] Gavin Andresen. *Bit-thereum.* http://gavintech.blogspot.co.uk/2014/06/bit-thereum.html. Last accessed: 3 June 2015.

[5] AnoniBet. *AnoniBet: Since 2011.* https://www.anonibet.com/default.aspx. Last accessed: 14 February 2015.

[6] Andreas M Antonopoulos. *Mastering Bitcoin: Unlocking Digital Cryptocurrencies.* "O'Reilly Media, Inc.", 2014.

[7] Grad Base. *The 21st century resume.* http://www.gradba.se/. Last accessed: 20 February 2015.

[8] Basecamp.com. *Ruby Gem: LocalTime.* https://github.com/basecamp/local_time. Last accessed: 9 June 2015.

[9] BBC. *Quest for lost harddrive with £4m stored bitcoins.* http://www.bbc.co.uk/news/technology-25138627. Last accessed: 29 January 2015.

[10] Satoshi Bet. *Fair Bitcoin Casino.* https://satoshibet.com/. Last accessed: 20 February 2015.

[11] BetBTC. *The Unique Bitcoin Sports Betting Exchange.* https://www.betbtc.co/. Last accessed: 14 February 2015.

[12] Betdaq.com. *Betdaq Betting Exchange.* http://www.betdaq.com/. Last accessed: 8 February 2015.

[13] Betfair.com. *Betfair Betting Exchange.* http://www.betfair.com/exchange. Last accessed: 8 February 2015.

[14] Betfair.com. *Betfair Rules and Regulations.* www.betfair.com/www/GBR/en/aboutUs/Rules.and.Regulations/. Last accessed: 8 February 2015.

[15] BetMoose. *Bet on Anything with Bitcoin.* https://www.betmoose.com/. Last accessed: 14 February 2015.

[16] BitBet. *BitBet.us Got Milk?* https://bitbet.us/. Last accessed: 14 February 2015.

[17] BitcoinAverage.com. *BitcoinAverage Integration API.* https://bitcoinaverage.com/api. Last accessed: 9 June 2015.

[18] Bitcoin.org. *Bitcoin Core.* https://github.com/bitcoin/bitcoin. Last accessed: 6 February 2015.

[19] Bitcoin.org. *Developer Guide: Block chain.* https://bitcoin.org/en/developer-guide#block-chain. Last accessed: 29 January 2015.

[20] Bitcoin.org. *Developer Guide: Block chain overview.* https://bitcoin.org/en/developer-guide#block-chain-overview. Last accessed: 28 January 2015.

[21] Bitcointalk.org. *BitcoinTalk Forum.* https://bitcointalk.org/. Last accessed: 11 June 2015.

[22] Bitmessage.org. *Bitmessage: A Peer to Peer Message Authentication and Delivery System.* https://bitmessage.org/bitmessage.pdf. Last accessed: 6 February 2015.

[23] Kevin Blackwood. *Casino gambling for dummies.* John Wiley & Sons, 2011.

[24] Blockchain.info. *Hash Rate Chart.* https://blockchain.info/charts/hash-rate. Last accessed: 29 January 2015.

[25] BlockChain.info. *Market Capitalization.* https://blockchain.info/charts/market-cap. Last accessed: 24 January 2015.

[26] BlockChain.info. *Total Number of Transactions.* https://blockchain.info/charts/n-transactions-total. Last accessed: 24 January 2015.

[27] Blocktrail.com. *Powering Bitcoin Apps: Bitcoin API for developers and enterprise.* https://www.blocktrail.com/. Last accessed: 30 May 2015.

[28] Bjorn Blomqvist. *Ruby Gem: QRCodeWithPatches.* https://github.com/bjornblomqvist/rqrcode. Last accessed: 9 June 2015.

[29] BrakemanScanner.org. *Static analysis security scanner for Ruby on Rails.* http://brakemanscanner.org/. Last accessed: 11 June 2015.

[30] BTCrow.com. *BTCrow.com: The Bitcoin Escrow Service.* https://btcrow.com/. Last accessed: 3 June 2015.

[31] Bytecoin.org. *Bytecoin.* http://www.bytecoin.org. Last accessed: 24 January 2015.

[32] Scott Campbell. *Bitcoin exchange MtGox faced 150000 hack attacks every second.* http://www.telegraph.co.uk/finance/currency/10686698/Bitcoin-exchange-MtGox-faced-150000-hack-attacks-every-second.html. Last accessed: 22 May 2015.

[33] Chain.com. *Connect to the Financial Cloud.* https://chain.com/. Last accessed: 11 June 2015.

[34] Chain.com. *Ruby Gem: ChainRuby.* https://github.com/chain-engineering/chain-ruby. Last accessed: 11 June 2015.

[35] Chain.so. *SoChain's fast blockchain API is the easiest, most cost-effective way to build applications on Dogecoin, Bitcoin, and Litecoin.* https://chain.so/api. Last accessed: 11 June 2015.

[36] G. Charlton. *EConsultancy: U.K Online Gambling Sector.* https://econsultancy.com/blog/62407-uk-s-online-gambling-sector-worth-2bn-in-2012-stats/. Last accessed: 24 January 2015.

[37] Norman Clarke. *Ruby Gem: FriendlyId.* https://github.com/norman/friendly_id. Last accessed: 9 June 2015.

[38] C. Cohen and N. Forrester. *Beenz.com.* http://www.beenz.com, 1998. Last accessed: 24 January 2015.

[39] Coinprism. *80 bytes OP_RETURN explained.* http://blog.coinprism.com/2015/02/11/80-bytes-op-return/. Last accessed: 30 May 2015.

[40] International Cricket Council. *Live Cricket Scores and News.* http://www.icc-cricket.com/. Last accessed: 30 May 2015.

[41] Counterparty.io. *Counterparty: Financial Tools on the Bitcoin Network.* http://counterparty.io/. Last accessed: 7 February 2015.

[42] CryptoNote.org. *CryptoNote: an open-source technology.* https://cryptonote.org/. Last accessed: 6 February 2015.

[43] Curecoin. *The Cure for Common Crypto-currency.* https://www.curecoin.net/. Last accessed: 6 February 2015.

[44] Satoshi Dice. *The Original Blockchain-based Bitcoin Casino Dice Game.* https://satoshidice.com/. Last accessed: 20 February 2015.

[45] DirectBet. *DirectBet Twitter Account.* https://twitter.com/DirectBetEU. Last accessed: 14 February 2015.

[46] DirectBet. *Sports Betting with Bitcoins, Litecoins, Dogecoins & Darkcoins.* http://www.directbet.eu/. Last accessed: 14 February 2015.

[47] Directory.io. *The Bitcoin private key database.* http://directory.io/. Last accessed: 7 June 2015.

[48] Ethereum.org. *A Next Generation Smart Contract and Decentralized Application Platform.* https://www.ethereum.org/pdfs/EthereumWhitePaper.pdf. Last accessed: 7 February 2015.

[49] Ethereum.org. *Ethereum Website.* https://www.ethereum.org/. Last accessed: 7 February 2015.

[50] TP Faucet. *TP's TestNet Faucet.* http://tpfaucet.appspot.com/. Last accessed: 25 May 2015.

[51] Bitcoin Forum. *Bitcoin Gambling Forum.* https://www.bitcoinforum.com/gambling/. Last accessed: 22 May 2015.

[52] Bitcoin Foundation. *Bitcoin Foundation: What We Do.* http://bitcoinfoundation.org/. Last accessed: 3 June 2015.

[53] Inc. GitHub. *GitHub is the best place to share code with friends, co-workers, classmates, and complete strangers. Over eight million people use GitHub to build amazing things together.* https://github.com/about. Last accessed: 9 June 2015.

[54] Mauricio Gomes. *Ruby Gem: Api_Auth.* https://github.com/mgomes/api_auth. Last accessed: 11 June 2015.

[55] Google. *Google Analytics:Turn insights into action.* https://www.google.com/analytics/. Last accessed: 11 June 2015.

[56] UK Government. *2005 Gambling Act.* http://www.legislation.gov.uk/ukpga/2005/19/contents. Last accessed: 8 February 2015.

[57] UK Government. *Gambling Commission.* http://www.gamblingcommission.gov.uk/Home.aspx. Last accessed: 8 February 2015.

[58] Andy Greenberg. *FBI Says It's Seized $28.5 Million In Bitcoins From Ross Ulbricht, Alleged Owner of Silk Road.* http://www.forbes.com/sites/andygreenberg/2013/10/25/fbi-says-its-seized-20-million-in-bitcoins-from-ross-ulbricht-alleged-owner-of-silk-road/. Last accessed: 25 May 2015.

[59] The PostgreSQL Development Group. *The world's most advanced open-source database.* http://www.postgresql.org. Last accessed: 9 June 2015.

[60] Heroku.com. *Announcing Better SSL For Your App.* https://blog.heroku.com/archives/2012/5/3/announcing_better_ssl_for_your_app. Last accessed: 11 June 2015.

[61] James Hudson. *UK sports betting market continues to grow driven by gen Y and mobile betting apps.* http://www.favourit.com/en/content/uk-sports-betting-stats-mobile. Last accessed: 7 February 2015.

[62] ibas uk.com. *2007 Annual Report.* http://www.ibas-uk.com/pdf/IBAS_Annual%20Report_07.pdf. Last accessed: 8 February 2015.

[63] ibas uk.com. *Independent Betting Adjudication Service.* http://www.ibas-uk.com/. Last accessed: 8 February 2015.

[64] Spike Ilacqua. *Ruby Gem: Strongbox.* https://github.com/spikex/strongbox. Last accessed: 11 June 2015.

[65] Amazon Web Services Inc. *Amazon Elastic Compute Cloud (Amazon EC2) is a web service that provides resizable compute capacity in the cloud.* http://aws.amazon.com/ec2/. Last accessed: 9 June 2015.

[66] Heroku Inc. *A platform as a service (PaaS) that enables developers to build and run applications entirely in the cloud.* https://dashboard.heroku.com/. Last accessed: 9 June 2015.

[67] The Tor Project Inc. *Tor: Anonymity Online.* https://www.torproject.org/index.html. Last accessed: 25 May 2015.

[68] InterCasino. *InterCasino History.* https://www.intercasino.com/en/about. Last accessed: 24 January 2015.

[69] D. Jackson and B. Downey. *E-Gold Ltd.* http://www.e-gold.com, 1996. Last accessed: 24 January 2015.

[70] Reality Keys. *Facts about the future, cryptographic proof when they come true.* https://www.realitykeys.com//. Last accessed: 20 February 2015.

[71] Kickstarter.com. *Ruby Gem: Rack-Attack.* https://github.com/kickstarter/rack-attack. Last accessed: 11 June 2015.

[72] AlterEGO Labs. *Ruby Gem: BetfairApiNgRails.* https://github.com/alterego-labs/betfair_api_ng_rails. Last accessed: 9 June 2015.

[73] Litecoin.org. *Litecoin.* https://www.litecoin.org. Last accessed: 24 January 2015.

[74] Lotology.co.uk. *40 Million in Winnings.* http://lotology.co.uk/40-million-scratchcard-winnings/. Last accessed: 7 February 2015.

[75] National Lottery. *The UK National Lottery.* https://www.national-lottery.co.uk/. Last accessed: 7 February 2015.

[76] Mastercoin.org. *Mastercoin Spec.* https://github.com/mastercoin-MSC/spec. Last accessed: 7 February 2015.

[77] John Mettraux. *Ruby Gem: Rufus-Scheduler.* https://github.com/jmettraux/rufus-scheduler. Last accessed: 9 June 2015.

[78] Erik Michaels-Ober. *Ruby Gem: BitcoinAverage.* https://github.com/sferik/bitcoinaverage. Last accessed: 9 June 2015.

[79] David Zeiler Money Morning. *Why Virtual Currency is Here to Stay - Bitcoin or No Bitcoin.* http://moneymorning.com/2013/11/22/why-virtual-currency-is-here-to-stay-bitcoin-or-no-bitcoin/. Last accessed: 24 January 2015.

[80] S. Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System.* 2008. https://bitcoin.org/bitcoin.pdf, Last accessed: 24 January 2015.

[81] BBC News. *Liberty Reserve: Barclays aiding money-laundering probe.* http://www.bbc.co.uk/news/business-22746768. Last accessed: 25 May 2015.

[82] CBC News. *Mt. Gox shutdown a major blow for bitcoin.* http://www.cbc.ca/news/technology/mt-gox-shutdown-a-major-blow-for-bitcoin-1.2550256. Last accessed: 25 May 2015.

[83] Cryptocoin News. *Bitcoin 2.0.* https://www.cryptocoinsnews.com/bitcoin-2-0-will-big-deal/. Last accessed: 24 January 2015.

[84] NitrogenSports. *Changing the way the world bets on sports.* https://nitrogensports.eu/. Last accessed: 14 February 2015.

[85] Proof of Existence. *Select a document and have it certified in the Bitcoin blockchain.* http://www.proofofexistence.com/. Last accessed: 20 February 2015.

[86] Massachusetts Institute of Technology. *MIT PGP Public Key Server.* https://pgp.mit.edu/. Last accessed: 29 May 2015.

[87] Association of Tennis Professionals. *ATP World Tour - Official Site of Men's Professional Tennis.* http://www.atpworldtour.com/. Last accessed: 30 May 2015.

[88] Organ Ofcorti. *Estimating the Number of Bitcoin Miners.* http://organofcorti.blogspot.co.uk/2014/05/165-estimating-number-of-bitcoin-miners.html. Last accessed: 27 January 2015.

[89] Christoph Olszowka. *Ruby Gem: Simplecov.* https://github.com/colszowka/simplecov. Last accessed: 11 June 2015.

[90] Online-Convert.com. *Calculate a SHA hash with 256 bits.* http://hash.online-convert.com/sha256-generator. Last accessed: 30 May 2015.

[91] Opidoki. *Oracle Programming Interface for Counterparty.* https://github.com/brighton36/opidoki. Last accessed: 20 February 2015.

[92] Orisi. *Distributed Bitcoin Oracles.* http://orisi.org/. Last accessed: 20 February 2015.

[93] Charlie Osborne. *Bitstamp exchange reopens doors after $5m hack.* http://www.zdnet.com/article/bitstamp-exchange-reopens-doors-after-5m-hack/. Last accessed: 22 May 2015.

[94] Palkeo.com. *How to Steal Bitcoins.* http://www.palkeo.com/code/stealing-bitcoin.html. Last accessed: 7 June 2015.

[95] Santiago Pastorino and Carlos Antonio da Silva. *Ruby Gem: Rails-Api.* https://github.com/rails-api/rails-api. Last accessed: 11 June 2015.

[96] Abraham Polishchuk. *Building a RESTful API in a Rails Application.* https://www.airpair.com/ruby-on-rails/posts/building-a-restful-api-in-a-rails-application. Last accessed: 11 June 2015.

[97] Betfair Developer Program. *Betfair Exchange API.* https://developer.betfair.com/. Last accessed: 9 June 2015.

[98] Reddit.com. *The front page of the Internet.* https://www.reddit.com/. Last accessed: 11 June 2015.

[99] RichOnRails. *Basic AJAX in Ruby on Rails.* https://richonrails.com/articles/basic-ajax-in-ruby-on-rails. Last accessed: 9 June 2015.

[100] I Nelson Rose. *Gambling and the Law.* Gambling Times Los Angeles, 1986.

[101] Margaret Rouse. *Dictionary Attack Definition.* `http://searchsecurity.techtarget.com/definition/dictionary-attack`. Last accessed: 7 June 2015.

[102] RSpec.info. *Behaviour Driven Development for Ruby. Making TDD Productive and Fun.* `http://rspec.info/`. Last accessed: 11 June 2015.

[103] Rspec.info. *Ruby Gem: Rspec-Rails.* `https://github.com/rspec/rspec-rails`. Last accessed: 11 June 2015.

[104] RubyDoc.org. *Ruby BigDecimal.* `http://ruby-doc.org/stdlib-1.9.3/libdoc/bigdecimal/rdoc/BigDecimal.html`. Last accessed: 11 June 2015.

[105] RubyOnRails.org. *Active Record Query Interface.* `http://guides.rubyonrails.org/active_record_querying.html`. Last accessed: 9 June 2015.

[106] rubyonrails.org. *Ruby on Rails is an open-source web framework that's optimized for programmer happiness and sustainable productivity. It lets you write beautiful code by favoring convention over configuration.* `http://rubyonrails.org/`. Last accessed: 9 June 2015.

[107] Katherine Rushton. *Liberty Reserve shut down in $6bn money laundering case.* `http://www.telegraph.co.uk/finance/10085600/Liberty-Reserve-shut-down-in-6bn-money-laundering-case.html`. Last accessed: 25 May 2015.

[108] ScanMyServer.com. *Test the Security of Your Website or Blog, Free.* `https://www.scanmyserver.com/`. Last accessed: 11 June 2015.

[109] Matthew Sparkes. *The £625m lost forever - the phenomenon of disappearing Bitcoins.* `http://www.telegraph.co.uk/technology/news/11362827/The-625m-lost-forever-the-phenomenon-of-disappearing-Bitcoins.html`. Last accessed: 7 June 2015.

[110] Bitcoin Sportsbook. *8 Bitcoin Betting Scams and Lessons to Learn.* `https://www.bitcoinsportsbooks.com/blog/8-bitcoin-betting-scams-lessons-to-learn/`. Last accessed: 22 May 2015.

[111] The Telegraph. *Bahrain Olympic champion Rashid Ramzi in positive drugs test.* `http://www.telegraph.co.uk/sport/othersports/drugsinsport/5243525/Bahrain-Olympic-champion-Rashid-Ramzi-in-positive-drugs-test.html`. Last accessed: 24 January 2015.

[112] Test.Webbtc.com. *The Testnet 3 API.* `http://test.webbtc.com/api`. Last accessed: 11 June 2015.

[113] Bitcoin Trezor. *The Bitcoin Safe.* `https://www.bitcointrezor.com/`. Last accessed: 29 January 2015.

[114] Twitter. *Twitter Bootstrap: get bootstrap.* `http://getbootstrap.com/`. Last accessed: 9 June 2015.

[115] Stanford University. *Folding at Home.* `http://folding.stanford.edu/`. Last accessed: 6 February 2015.

[116] Lian Uphnix. *Ruby Gem: BitcoinRuby.* `https://github.com/lian/bitcoin-ruby`. Last accessed: 11 June 2015.

[117] Bitcoin Wiki. *BIP 0021: URI scheme.* `https://en.bitcoin.it/wiki/BIP_0021`. Last accessed: 9 June 2015.

[118] Bitcoin Wiki. *Brainwallet.* `https://en.bitcoin.it/wiki/Brainwallet`. Last accessed: 7 June 2015.

[119] Bitcoin Wiki. *Confirmation.* `https://en.bitcoin.it/wiki/Confirmation`. Last accessed: 29 January 2015.

[120] Bitcoin Wiki. *Contracts.* `https://en.bitcoin.it/wiki/Contracts`. Last accessed: 2 June 2015.

[121] Bitcoin Wiki. *Contracts: Using External State.* `https://en.bitcoin.it/wiki/Contracts#Example_4%3a_Using_external_state`. Last accessed: 3 June 2015.

[122] Bitcoin Wiki. *Controlled Supply.* `https://en.bitcoin.it/wiki/Controlled_supply`. Last accessed: 29 January 2015.

[123] Bitcoin Wiki. *Promotional Graphics.* `https://en.bitcoin.it/wiki/Promotional_graphics`. Last accessed: 6 February 2015.

[124] Bitcoin Wiki. *Proof of Burn.* `https://en.bitcoin.it/wiki/Proof_of_burn`. Last accessed: 7 February 2015.

[125] Bitcoin Wiki. *Target.* `https://en.bitcoin.it/wiki/Target`. Last accessed: 29 January 2015.

[126] Bitcoin Wiki. *Testnet.* `https://en.bitcoin.it/wiki/Testnet`. Last accessed: 25 May 2015.

[127] Bitcoin Wiki. *Weaknesses.* `https://en.bitcoin.it/wiki/Weaknesses`. Last accessed: 29 January 2015.

[128] Wikipedia.org. *Gambling.* `http://en.wikipedia.org/wiki/Gambling`. Last accessed: 7 February 2015.

[129] C. Chambers-Jones with H. Hillman. *Financial Crime and Gambling in a Virtual World.* Edward Elgar Publishing Limited, 2014. ISBN: 978 1 78254 520 0.

[130] Wopata.com. *Ruby Gem: wkhtmltopdf-binary.* `https://github.com/wopata/wkhtmltopdf-binary`. Last accessed: 9 June 2015.