Micropayments for Decentralized Currencies

Rafael Pass*
Cornell University
rafael@cs.cornell.com

abhi shelat†
U of Virginia
abhi@virginia.edu

February 18, 2016

Abstract

Electronic financial transactions in the US, even those enabled by Bitcoin, have relatively high transaction costs. As a result, it becomes infeasible to make *micropayments*, i.e. payments that are pennies or fractions of a penny.

To circumvent the cost of recording all transactions, Wheeler (1996) and Rivest (1997) suggested the notion of a *probabilistic payment*, that is, one implements payments that have *expected* value on the order of micro pennies by running an appropriately biased lottery for a larger payment. While there have been quite a few proposed solutions to such lottery-based micropayment schemes, all these solutions rely on a trusted third party to coordinate the transactions; furthermore, to implement these systems in today's economy would require a a global change to how either banks or electronic payment companies (e.g., Visa and Mastercard) handle transactions.

We put forth a new lottery-based micropayment scheme for any ledger-based transaction system, that can be used today without any change to the current infrastructure. We implement our scheme in a sample web application and show how a single server can handle thousands of micropayment requests per second. We analyze how the scheme can work at Internet scale.

1 Introduction

This paper considers methods for transacting very small amounts such as $\frac{1}{10}^{\text{th}}$ to 1 penny. Traditional bank-based transactions usually incur fees of between 21 to 25 cents (in the US) plus a percentage of the transaction [16] and thus transactions that are less than 1$ are rare because of this inefficiency; credit-card based transactions can be more expensive.

*Supported in part by NSF Award CNS-1217821, NSF Award CCF-1214844 and AFOSR Award FA9550-15-1-0262.

†Supported by NSF Award CNS-0845811, TC-1111781, and the Microsoft Faculty Fellowship.

Although several new crypto-currencies have removed the centralized trust from a currency and have substantially reduced the cost of a large *international* transaction, they have not solved the problem of reducing transaction fees to enable micro-payments. In Fig. 1, we show that Bitcoin transaction fees are usually at least 0.0001 bitcoin, which corresponds to between 2.5 and 10 cents over the last two years. See Fig. 8 in the Appendix for another graph showing the distribution of fees among recent transactions.

The transaction fee pays for the *cost of bookkeeping*, *credit risk* and *overhead due to fraud.* Although the cost of storage and processing have diminished, the cost of maintaining reliable infrastructure for transaction logs is still noticeable.

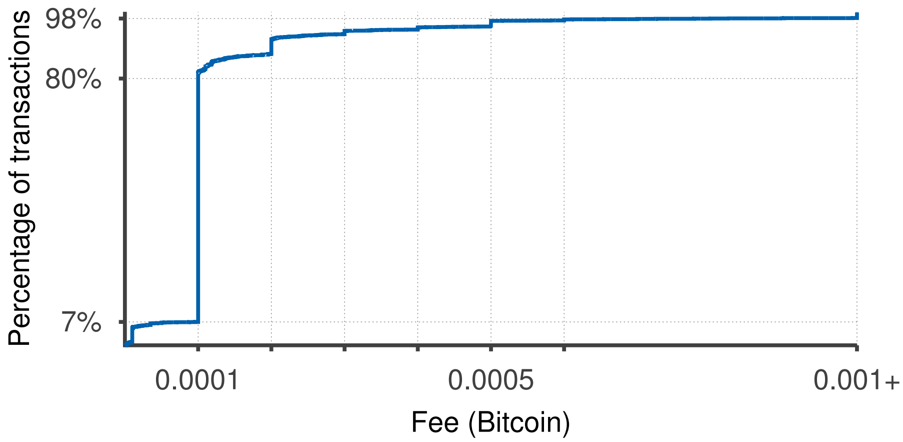

Figure 1: A plot of transaction fee versus frequency for 1 million transactions in May 2015. Very few transactions have fees less than 0.0001 Bitcoin. As of May 2015, 10k milliBitcoin, or 0.0001 bitcoin corresponds to roughly 2.5 cents.

One method for overcoming a transaction fee is to *batch* several small transactions for a user into a large transaction that occurs say, monthly. Standard implementations of this idea, however, rely on the extension of *credit* to the user from a merchant or bank, and thus, incurr credit risk. Systems like Apple iTunes and Google play apparently implement their $1 transactions using a probabilistic model for user behavior to pick an optimal time to balance credit risk versus transaction fee. Systems like Starbucks attempt to sell pre-paid cards for which several orders result in one credit transaction. PayPal introduced a micro-payments pricing model (5.0% plus $0.05). Similarly, the Bitcoinj project (see `https://bitcoinj.github.io/working-with-micropayments`) enables setting up a micropayment channel to a *single* predetermined party (e.g., a single webpage): Each payer must set up a separate channel and escrow account for each merchant; moreover, the merchants require a bookkeeping system for each user (to issue a "claw-back" transactions). In contrast, we are here interested in a decentralized payment system where users can make micropayments to *anyone.*

Lottery-based Micropayments Wheeler [20] and Rivest [19] suggested a intriguing approach to overcome the cost of bookkeeping for small transactions. The idea in both works is to employ *probabilistic* "lottery-based" payments: to provide a payment of X, the payer issues a "lottery ticket" that pays, say, $100X$ with probability $\frac{1}{100}$. In expectation, the merchant thus receives $\frac{1}{100} \cdot 100X = X$, but now (in expectation) only 1 in a hundred transactions "succeeds", and thus the transaction cost becomes 100 times smaller. Several implementations of this idea subsequently appeared; most notable among them is the Peppercoin scheme by Micali and Rivest [15] which provided a convenient non-interactive solution.

However, these elegant ideas all require a *trusted third party*—either a bank or an electronic payment companies (e.g., Visa and Mastercard)—to coordinate the transactions. In this case, the trusted party cannot be verified or audited to ensure that it is performing its job correctly. Furthermore, to implement these systems in today's economy requires a *global* change to banks and/or electronic payment companies that handle transactions. Consequently, such solution have gained little traction in real-life system.

Cryptocurrency-based Micropayments In this paper, we propose micropayment systems based on cryptocurrencies. We follow the lottery-based approach put forth by Wheeler [20] and Rivest [19] and show how to implement such an approach using any suitable crypto-currency system. We provide two main solutions:

- Using the current Bitcoin/altcoin scripting language, we provide an implementation of lottery-based micropayments that only relies on a *publicly-verifiable* third party; that is, anyone can verify that the third party is correctly fulfilling its proper actions. This solution also enables performing transaction with fast validation times (recall that standard Bitcoin transactions require roughly 10 minute validations, which is undesirable in the context of micropayments). Using this solutions, bitcoin-based micropayments can be implemented today without *any change* to the current infrastructure.

- We also suggest simple modifications to the Bitcoin scripting language that enables implementing lottery-based micropayments without the intervention of any third party. Furthermore, this scheme can be directly implemented in the Ethereum currency [7] without any modification to the scripting language. (Validation times for transaction, however, are no longer faster than in the underlying cryptocurrency.)

At a high-level, the idea behind our solution is the following: The user starts by transferring $100X$ into an "escrow". This escrow transaction has an associated "puzzle", and *anyone* that has a solution to this puzzle can spend the escrow. Roughly speaking, the solution to the puzzle consists of a signed transcript of a cryptographic coin-tossing protocol (where the signature is with respect to to the user's public key) such that the string computed in the coin-tossing ends with 00 (an event that happens with probability 1/100 by the security of the coin-tossing protocol).

Whenever the payer wants to spend X, it engages with a merchant in a coin-tossing protocol and agrees to sign the transcript. The merchant thus receives a signed coin-tossing transcript in every transaction, and additionally, with probability 1/100, the coin-tossing transcript yields a solution to the puzzle (i.e., the string computed in the coin-tossing protocol ends with 00). The merchant can thus spend the money (i.e., $100X$) placed in escrow.

This approach, which we refer to as MICROPAY1, however, cannot be directly implemented today because of limitations in crypto-currency scripting languages. Additionally, validation times for Bitcoin transactions are high and a malicious spender who issues several transactions in parallel can cheat.

Our solution, MICROPAY2, makes use of a *verifiable* trusted third party—which we refer to as a *Verifiable Transaction Service (VTS)*—to overcome these issues. Roughly speaking, the VTS performs a specific polynomial-time computation and signs certain messages in case the computations produce a desired result: in our case, the VTS checks whether a coin-tossing transcript is winning, and if so it "releases" the escrow by signing a release transaction. Thus, anyone can verify that the VTS only signs messages correctly (by checking that the computation indeed gave the desired result). Furthermore, the VTS is only invoked on winning transactions (i.e., on average every 1/100 transactions) and can thus handle a large volume of transactions. Since the VTS only agrees to sign the escrow release *once*, MICROPAY2 implements fast transaction validation times. That is, merchants can be assured that as long as the VTS is acting honestly, as soon as they receive a signature from the VTS, they will receive their payment without having to wait for the transaction to appear on the block-chain. Furthermore, if the VTS is acting dishonestly (i.e., if it signs multiple times), this will be observed. Using a standard approach with locktime, our protocol can also be slightly modified to ensure that the user can always recover its money from the escrow within some pre-determined expiration time. MICROPAY2 also implements a penalty mechanism to force sequential behaviour by the payer which prevents a large class of spending attacks.

Finally, MICROPAY2 can be modified into a solution called MICROPAY3 where the VTS never needs to be activated if users are honest—i.e., it is an *invisible* third party. This solution, however, has faster validation times than the underlying cryptocurrency only if one assumes that users are rational.

Generalization to "Smart-Contracts" We mention that our solution provides a *general* method for a user A to pay x to different user B if some pre-determined polynomial-time computation produces some specific output (in the micropayment case, the polynomial time computation is simply checking whether the most significant two bits of the random tape are 00.)

Projects like Ethereum [7] provide Turing-complete scripting languages for crypto-currencies. These systems require a much more sophisticated mechanism to evaluate the scripts associated with transactions in order to prevent attacks. Our methods enable extending these "smart-contract" to deal with *probabilistic events* (such as our micro-payment "lottery-tickets"). Furthermore, we enable using other *current* cryptocurrencies (such as Bitcoin) to implement a large class of "smart-contracts" even if the contract may be written in a more complex language than what is currently supported by the scripting languages for the currency. Finally, our method enables using *soft* contracts, where the polynomial-time processes that determines if A should pay x to B may take as inputs also facts about the world (e.g., the whether the "Red Sox beat Yankees" in a particular game), or the process may even be specified in natural language, as long as the outcome of the process is publicly verifiable.

Applications of our Micropayment System We outline some applications that may be enabled by our system. We emphasize that none of these applications require any changes to current transactional infrastructures. To make these applications feasible however, it is critical that the user only needs to setup once, and be able to interact with any number of merchants, as opposed, to say, a "channels" system which requires the user to perform a different escrow transaction with each merchant.

An Ad-Free Internet: Our micropayment system could be used to replace advertisements on the Internet. Users can request an "ad-free version" of a webpage by using the protocol `httpb://` (instead of `http://`) which transparently invokes our micropayment protocol and then serves a page instead of having the server display an ad on the requested page. In Section 4, we report on an implementation of this idea.

Pay-as-you-go Games and Services: Our micropayment system could be used to enable pay-as-you go WiFi internet connections where users pay for every packet they send. Internet content providers (e.g., newspapers, magazines, blogs, music and video providers) and game-writers could charge for every item requested by the user, or for game-playing by the minute.

Generalized wagering In some of our schemes, a trusted party is used to sign a message if a certain event occurs. In our case, the event relates to a coin-tossing protocol that is executed between two parties. In general, one can imagine that the trusted-party signs statements about worldly events that have occurred such as "Red Sox beat Yankees" or "Patriots win Super Bowl", or interpret the outcome of contracts written in natural language. Using such a party, our protocols can be generalized to enable wagers that are implemented entirely without a bookkeeper, and only require the parties to trust a 3rd party who can digitally sign facts that can be publicly verified.

1.1 Prior work

Electronic payments research is vast. This work follows a series of paper [20, 19, 12, 15] on the idea of probabilistic payments, and improves upon them by removing or simplifying the trust assumptions and bootstrap requirements for their systems by using a crypto-currency, by simplifying the cryptographic assumptions needed, and by demonstrating a practical system in a web-server that implements the protocol. Some prior works focus on reducing the number of digital signatures required by the protocol: this concern is no longer a bottleneck. Moreover, none of those schemes focus on how to implement the transfer (they all require a bank to handle it).

An older form of digital currency is studied in [6, 5, 13]. These schemes rely on digital signatures from a trusted-third party (such as a bank) to record transfer of ownership of a coin. Various (online and off-line) methods are considered to prevent double-spending attacks. The schemes are not optimized for handling micropayments, and the economics of the scheme do not depart from the economics of current credit-card or ACH network based transactions. In some cases, the schemes offer a level of anonymity not provided by credit-cards etc.

Coupon-based schemes [10, 1, 18] are similar and require a trusted-party to issue coupons to users, then users spend these coupons with merchants, who then return the coupon to the trusted-party. The main focus for this line of research was to optimize the cryptographic operations that were necessary; today, these concerns are not relevant as we show in our evaluation section (see §4). Furthermore, these schemes have double-spending problems and require a trusted-party to broker all transactions and issue and collect coupons.

A few recent works discuss lotteries and Bitcoin, but none focus on reducing transaction costs or allowing a single setup to issue micropayments to an unlimited number of merchants. Andrychowicz *et al.* [2] implement Bitcoin lotteries using $O(n)$ or $O(n^2)$ ledger transactions *per lottery* where n is the number of players. Bentov and Kumaresan [3] discuss UC modeling and achieving fairness in secure computation by providing an abstract description of how to enforce penalties with Bitcoin through a novel "ladder mechanism" that uses $O(n)$ transactions per penalty. In contrast, the main idea in our work is to amortize 2-3 transaction fees over thousands of lottery protocol instances.

The goal of Mixcoin [4] is anonymity, and with this different motivation (see its footnote 12), the paper describes how to charge for mixing in a probabilistic way. Their mechanism differs in that it uses a random beacon, i.e. a public trusted source of randomness for the lottery, which does not work for micropayments.

As mentioned, the Bitcoinj project (see `https://bitcoinj.github.io/working-with-micropayments`) enables setting up a micropayment channel to a *single* predetermined merchant (e.g., a single webpage), by establishing a new address for the merchant, so this scheme falls short of our goal of one decentralized payment system where users can make micropayments to anyone.

The most comparable scheme to ours is the Lightning network [17] which users a network of such channels to enable payments between two arbitrary parties while only requiring each party to setup a small number of channels. The Lightning network does not use a trusted party, but rather, each pair of parties posts a transaction to the network in which both have contributed an equal amount, say 100 each. Through (off-chain) private communication, the pair can exchange transactions, which if posted to the network, would spend α and $(200 - \alpha)$ to addresses chosen by each party respectively. To maintain soundness of such a scheme, the network proposes the concept of Revocable Sequence Verification Contracs, which require several layers of transactions, penalties, and soft forks of the bitcoin protocol to support sequence majority. Finally, through the notion of a "Hashed timelock," a micropayment can be relayed from Alice to Dave via intermediaries, albeit via several RSVCs between the parties. In contrast, schemes in this paper use a verifiable trusted party and a single penalty mechanism to enforce soundness.

1.2 Outline of the paper

In Section 2 we provide a detailed description of our protocol in an abstract crypto-currency scheme. This model leaves out many of the implementation details behind the currency protocol but enables describing our solution in a convenient way; in essence, this abstract model captures the principles underlying all modern ledger-based transactional systems (such as bitcoin and all alt-coins). In Section 3 we next describe how to implement the abstract solution using the actual Bitcoin scripting language and formalism. In Section 4 we describe our implementation and present experiments to demonstrate the practical feasibility of our MICROPAY2 solution. In particular, we report on the above mentioned "ad-free internet" application.

2 Protocols

Abstract Model for Crypto-currencies A cryptocurrency system implements a distributed *ledger* specifying how coins are transferred; we here ignore how miners are incentivized to ensure that the ledger is available and not manipulated, but instead describe how coins are transferred. *Very* roughly speaking, transactions are associated with a public-key pk and a "release condition" Π. A *transaction* from an address $a_1 = (pk, \Pi)$ to an address $a_2 = (pk', \Pi')$ is valid if it specifies some input x that satisfies the release condition Π, when applied to both to a_1 and a_2; that is $\Pi(x, a_1, a_2) = 1$. The most "standard" release condition Π^{std} is one where a transaction is approved when x is a signature with respect to the public key pk on a_2; that is, pk is a public-key for a signature scheme, the "owner" of the address has the secret key for this signature scheme (w.r.t. pk), and anyone with the secret key for this signature scheme can transfer bitcoins from the address by signing the destination address. The bitcoin protocol specifies a restrictive script language for describing the release condition Π (see §3 for more details on this script language); other crypto-currencies such as Ethereum [7] allow more expressive script languages. For exposition, we may ignore the concrete formalism of the scripting language and instead describe our solutions in prose.

Finally, to analyze our protocols, we assume that every valid transaction broadcast to the cryptocurrency network has some chance of being incorporated into the ledger in the next block. This is a *strong* assumption about fair access to the cryptocurrency ledger, in practice, it may be possible for the network to prevent a particular user from *ever* adding any transactions to the ledger. Alternatively, all of the miners in a network may decide not to confirm payments with release conditions that arise from our protocol.

2.1 MICROPAY 1

We first propose a strawman solution to micropayments that illustrates the main idea. This solution uses a release condition Π that does not require any third party at all, but has drawbacks that we discuss and then address in the next protocol. The only cryptographic primitive required by this protocol (apart from digital signatures) is that of a *commitment scheme* (see [11] for details) which can be implemented with any hash operation such as SHA or RIPEMD; both are supported in most crypto-currency scripting languages.

Escrow Set-up: To initialize a "lottery-ticket", a user U with $a = (\mathsf{pk}, \Pi^{\mathsf{std}})$ containing $100X$ coins generates a new key-pair $(\mathsf{pk}^{\mathsf{esc}}, \mathsf{sk}^{\mathsf{esc}})$ and transfers the $100X$ coins to an *escrow* address $a^{\mathsf{esc}} = (\mathsf{pk}^{\mathsf{esc}}, \Pi^{\mathsf{esc}})$ (by signing (a, a^{esc}) using its key corresponding to pk). For easy of exposition, we defer specifying the release condition Π^{esc}.

Payment Request: Whenever a merchant M wants to request a payment of X from U, it picks a random number $r_1 \leftarrow \{0,1\}^{128}$, generates a commitment $c \leftarrow \mathbf{Com}(r_1; s)$ (where s represents the string that can be used to open/reveal the commitment), generates a new bitcoin address a_2 (to which the payment should be sent) and sends the pair (c, a_2) to the payer U.

Payment Issuance: To send a probabilistic payment of X, user U picks a random string r_2, creates a signature σ on c, r_2, a_2 (w.r.t. to $\mathsf{pk}^{\mathsf{esc}}$) and sends σ to the merchant. The merchant verifies that the signature is valid.

We now return to specifying the release condition Π^{esc}. Define $\Pi^{\mathsf{esc}}(x, a_{\mathsf{esc}}, a_2) = 1$ if and only if

1. x can be parsed as $x = (c, r_1, s, r_2, \sigma)$
2. $c = \mathbf{Com}(r_1; s)$,
3. σ is a valid signature on (c, r_2, a_2) with respect to the public key $\mathsf{pk}^{\mathsf{esc}}$ and
4. if the first 2 digits of $r_1 \oplus r_2$ are 00.

In other words, the merchant can ensure a transfer from the escrow address to a_2 happens if it correctly generated the commitment c (and knows the decommitment information r_1, s), and then sent c, a_2 to U; U agreed to the transaction (by providing a valid signature on c, r_2, a_2), *AND* the coin toss won the lottery using $r_1 \oplus r_2$ as randomness.

Security Analysis It can be shown using standard arguments that the "coin-tossing" $r_1 \oplus r_2$ cannot be biased (by more than a negligible amount) by either the merchant or the user (if the merchant can bias it, it can either break the binding property of the commitment, or forge a signature; if the user can bias it, it can break the hiding property of the commitment.) As a consequence, whenever the user agrees to a transaction, the merchant has a 1/100 (plus/minus a negligible amount) chance of receiving a witness which enables it to release the money in the escrow address. More precisely, the following properties hold:

- **[P1]** Consider some potentially malicious user that correctly signs a transaction with non-negligible probability. Then, conditioned on the event that the user produces an accepting signature on a transaction, the merchant receives a witness for the escrow address with probability at least 1/100 (minus a negligible amount) as long as the merchant honestly follows the protocol.
- **[P2]** Even if the merchant is arbitrarily malicious, it cannot receive a witness for the escrow address with probability higher than 1/100 (plus a negligible amount), as long as the user honestly follows the protocol.

Front-running attacks A subtle problem with MICROPAY1 is the issue of *front running*. A malicious payer may monitor the cryptocurrency network and listen for transactions from the merchant that attempt to spend her escrow. Before the merchant's transaction can be confirmed, a cheating payer can broadcast her own *valid* transaction to the network (which she can produce herself by simulating a merchant in the protocol) to attempt spending her own escrow first[1]. The cryptocurrency network sees two competing transactions for the same escrow, and the one that is confirmed depends on several random factors; i.e. the payer has initiated a race to spend. In the best case, this attack reduces the Merchant's *expected revenue* from a payment since some winning tickets may lose the transaction race. In the worst case, if all payers in the system are sophisticated and have more resources than the merchants, merchants may never get paid.

Parallel spend attack A front-running attack is a special case of a more general *parallel spending* attack. Our protocols assume that the payer *serializes* his purchases; an attacker who issues several payments *in parallel* can reduce the expected revenue of a lottery ticket because among the several potential lottery winners, only one ticket can spend the escrow. The front running attack described above is an example in which the cheating payer issues a payment *to itself* when he becomes aware of a merchant who has a winning ticket.

Mitigating these attacks Both of these payer attacks can be mitigated by a *penalty escrow*. Along with a payment escrow, the payer places a penalty

[1]We thank Bonneau for pointing out the importance of this attack.

amount that is much larger than the benefit she can expect to receive from a parallel spending attack[2]. Each payment lottery ticket can be bound to the address of a unique penalty escrow. The penalty escrow can be spent in one of two ways: (a) at a certain "locktime" in the future, the payer herself can spend her penalty escrow, (b) before that locktime, the escrow can be spent to an *invalid address* by anyone presenting two winning lottery tickets for the same payment escrow. Condition (b) encodes the "burning" of the penalty escrow; i.e., no merchant or payer can recover the escrow, rather, it is *lost* to the payer. This condition prevents colluding merchants who attempt to steal the penalty escrow.

If a payer now attempts to spend-in-parallel, there is a chance that two winning lottery tickets are created. In this case, the losing merchant can penalize the cheating payer by burning the penalty escrow using both its winning lottery ticket and the lottery ticket used by the winning merchant to spend the payment escrow.

Scripting MICROPAY1 employs a release condition Π^{esc} that uses two operations that currently are not supported in the bitcoin scripting language. First, while digital signatures are supported in the script language, the language only permits checking the validity of signatures on messages derived from the *current transaction* in a very specific way; the checksig operation does not directly allow signature verification on messages of the form that we use in the protocol. A second problem is that arithmetic operations can only be applied to 32-bit values. Additionally, the penalty escrow release condition require these operations.

2.2 MICROPAY2: Using a VTS

To overcome these issues, the MICROPAY2 scheme employs a (partially-trusted) third party T, i.e. a *Verifiable Transaction Service (VTS)*. T's only task is to verify certain simple computations and, if the computations are correct, to release a signature on a transaction. If T ever signs a transaction that corresponds to an incorrect computation, there is *irrefutable* evidence that (unless the signature scheme is broken) T "cheated" (or has been corrupted), and so T can be legally punished and/or replaced. (To achieve greater robustness against corruption of T, the task can be split among multiple parties $T_1, T_2, \dots T_n$ so that it suffices if a majority of them correctly check the computations.)

MICROPAY2 follows the structure of MICROPAY1 but employs a simpler release script $\tilde{\Pi}_2^{\mathsf{esc}}$. This new release condition requires two signatures on a transaction (a multi-signature), one from the user, and one from the trusted party T. Roughly speaking, U always provides M a signature on a transaction to spend the escrow, and in case of a winning ticket, T verifies that the lottery ticket wins and then provides a second signature on the transaction to spend the escrow to M. That is, $\tilde{\Pi}_2^x((\sigma_1, \sigma_2), a^x, a_2) = 1$ if and only if σ_1 is a signature

[2] In theory, an attacker can issue an *unlimited* number of transactions in parallel. In practice, however, a parallel attack is quickly discovered on the network, and so the maximum benefit to such an attack can be estimated.

of the transaction (a^x, a_2) with respect to pk^x *and* σ_2 is a signature of the transaction (a^x, a_2) with respect to pk^T, where pk^T is T's *permanent* public key. MICROPAY2 also incorporates the penalty escrow release conditions described in the previous section. In more details, the system involves the following steps:

- **Penalty Escrow Setup:** User U with an address $a = (pk, \Pi)$ generates a new key-pair $(\mathsf{pk}^{\mathsf{pen}}, \mathsf{sk}^{\mathsf{pen}})$ and transfers λX bitcoins to a penalty escrow $a^{\mathsf{pen}} = (\mathsf{pk}^{\mathsf{pen}}, \tilde{\Pi}_2^{\mathsf{pen}})$ by signing (a, a^{pen}) using its key corresponding to pk. Recall the release condition $\tilde{\Pi}_2^{\mathsf{pen}}(x, a_{\mathsf{pen}}, a_2) = 1$ if and only if the transaction has been signed by both U and T. Before broadcasting this transaction to the network, U and T exchange the following: T provides U a partially signed transaction (using its signing key) that can be spent by U after an expiration time e. This can be implemented using lockTime; U simply signs the partially-signed transaction after lockTime to recover escrow. It is assumed that the expiration e for a given escrow can be publicly determined.

- **Escrow Setup:** User U with an address $a = (pk, \Pi^{\mathsf{std}})$ generates a new key-pair $(\mathsf{pk}^{\mathsf{esc}}, \mathsf{sk}^{\mathsf{esc}})$ and transfers X bitcoins to an "escrow" address $a^{\mathsf{esc}} = (\mathsf{pk}^{\mathsf{esc}}, \tilde{\Pi}_2^{\mathsf{esc}})$ by signing (a, a^{esc}) using its key corresponding to pk. The escrow and penalty escrow addresses must be bound; for example, the penalty escrow can be index 0 and the payment escrow can be index 1 of the same transaction.

- **Payment Request:** This step is identical to the one in MICROPAY1: Whenever a merchant M wants to request a payment of $X/100$ from U, it picks a random number $r_1 \leftarrow \{0,1\}^{128}$, generates a commitment $c = \mathbf{Com}(r_1; s)$ (where s represents the string that can be used to open/reveal the commitment), generates a new bitcoin address a_2 (to which the payment should be sent) and sends the pair (c, a_2) to the payer U.

- **Payment Issuance:** If the user U agrees to send a probabilistic payment $X/100$, it picks a random string r_2, creates 1) a signature σ_1 on the transaction (a^{esc}, a_2), 2) a signature σ on $(c, r_2, a_2, a^{\mathsf{pen}})$ with respect to $\mathsf{pk}^{\mathsf{esc}}$), and 3) a partially-signed transaction σ_{pen} and the current block height that allows T to spend the penalty escrow to address 0 (i.e., an invalid address for which there does not exist a secret key) within k block confirmations from the current block height[3]. U sends $\sigma_1, \sigma, \sigma_{\mathsf{pen}}$ to the merchant M. The merchant verifies that the payment and penalty escrows have not been spent, that the penalty address is bound to the escrow address, that the escrow's expiration time is not within the next k blocks, and that all signatures are valid.

- **Merchant Response:** M sends U the witness (r_1, s). If merchant M received a winning lottery ticket, then M sends the triple $(x, a^{\mathsf{esc}}, a_2)$ to T.

[3] As is standard, the signed transaction σ_{pen} spends the penalty escrow amount *minus* ϵ which is reserved to pay the transaction fee.

T computes a signature σ_T on the transaction (a^{esc}, a_2) using public key pk^T and sends it to M if and only if $x = (c, r_1, s, r_2, \sigma)$, $c = \mathbf{Com}(r_1; s)$, σ is a valid signature on (c, r_2, a_2) w.r.t. $\mathsf{pk}^{\mathsf{esc}}$, and the last 2 digits of $r_1 \oplus r_2$ are 00.

Furthermore, T publishes the tuple x (either on its own bulletin board, on the blockchain, or some alt-chain). If T ever signs (a^{esc}, a_2) without having made public a "witness" x, it is deemed faulty.

Finally, once M has received the signature σ_T from T, then M can spend a^{esc} to address a_2 (which it controls) using σ_1, σ_T to satisfy the release condition $\tilde{\Pi}^{\mathsf{esc}}$.

- **Penalty:** If a merchant presents two winning lottery tickets for the same payment escrow a_{esc}, and a partially signed penalty transaction σ_{pen} that has a block timestamp within k of the current block height, then T also signs the penalty transaction and spends it to address 0. T also publishes the witness for the penalty transaction by publishing the two winning tickets for $(a_{\mathsf{esc}}, a_{\mathsf{pen}})$.

- **Escrow recovery:** After locktime, the payer can recover unspent penalty escrows. Alternatively, after a payment escrow has been spent by a merchant and sufficiently confirmed on the blockchain, the payer can ask T to sign a transaction allowing the payer to recover the penalty escrow early. A payer can always recover its own payment escrow by creating merchant transactions to herself until she creates a winning lottery ticket.

Security Analysis The following claims can be easily verified using standard cryptographic techniques:

- *If T acts honestly*, then properties **P1** and **P2** from Section 2.1 hold.

- *If T deviates* from its prescribed instructions, then (a) except with negligible probability, this can be publicly verified, and (b) the only damage it can create is to bias the probability that the escrow is released in an otherwise approved transaction.

By the second claim, T can never "steal" the escrow money. By cheating, it can only transfer the money to a merchant (even for a losing lottery ticket), but only to a merchant to whom the user agreed to issue a (micropayment) transaction. Additionally, by cheating, it can withhold a payment for a merchant. By the first claim, if T performs either of these (cheating) actions, this can be noticed.

Malicious Merchants A malicious (and not rational) merchant may attempt to incorrectly cause a payer's penalty escrow to be burned by receiving a winning ticket, not spending the ticket immediately, but instead waiting for the payer to interact with another merchant that receives a winning ticket, and then finally announcing the original winning ticket as evidence of double spending. Rational merchants would not implement this attack since they must forfeit the entire escrow amount in order to cause the payer to lose the penalty escrow. Nonetheless, to prevent this attack, the Merchant sends the Payer the witness (r_1, s) for all tickets. Thus, a payer immediately learns if the ticket has won (and therefore, the escrow cannot be spent again), or lost (in which case it is safe for the payer to continue spending). If a payer does not receive the witness for a transaction, the payer must wait until the penalty transaction expires (k blocks) before spending again.

Fast Validation Times We finally remark that if T only agrees to sign the escrow release *once*, MICROPAY2 implements fast transaction validation times. That is, merchants can be assured that as long as the T is acting honestly, as soon as they receive a signature from T, they will receive their payment (without having to wait for the transaction to appear on the block-chain). Furthermore, if the VTS is acting dishonestly (i.e., if it signs multiple times), this will be observed.

2.3 MICROPAY3: Using an "Invisible" VTS

MICROPAY2 requires the intervention of T in every winning transaction. In the next protocol, MICROPAY3, the VTS T is only invoked when either user or merchant deviates from their prescribed instructions. In this sense, the trusted third party T is *invisible* in the optimistic (honest) case. MICROPAY3, however, only implements faster validation time than Bitcoin assuming that the payers are rational (malicious payers can double spend, and are then penalized).

MICROPAY3 proceeds similarly to MICROPAY2, with the key difference that U releases the money to M whenever M receives a winning ticket by signing the transaction as T would have done.

If U is not willing to sign the payment escrow, M uses T to spend. Upon learning that M has won, if U attempts to race and spend its own escrow before M, then it must create a distinct signed winning tickets, and M can then invoke the penalty procedure with T using its own partially signed winning ticket as a witness for cheating. Together, in MICROPAY3, U and M's enjoy greater privacy of their micropayments since T need not be involved in the winning transactions in the optimistic case.

To implement this idea, the payment escrow release condition uses a 2-out-of-3 multi-signature under keys $\mathsf{pk}^{\mathsf{esc}}$ and pk^T as before, and also under another key $\mathsf{pk}_2^{\mathsf{esc}}$ generated by the user. That is, the new release condition is, $\widetilde{\Pi}_3^{\mathsf{esc}}((\sigma_1, \sigma_2), a^{\mathsf{esc}}, a_2) = 1$ if and only if σ_1 and σ_2 are unique signatures of the transaction (a^{esc}, a_2) with respect to two distinct keys among the following three keys: $\mathsf{pk}^{\mathsf{esc}}$, $\mathsf{pk}_2^{\mathsf{esc}}$, or pk^T. As before, to send a micropayment, U sends M a partially-signed transaction using key $\mathsf{pk}^{\mathsf{esc}}$. Now, if the ticket wins, then M *first asks* U for another signature on the same transaction under $\mathsf{pk}_2^{\mathsf{esc}}$. Using these two signatures, M can satisfy the release condition and spend the escrow.

- **Penalty escrow:** The setup is the same as MICROPAY2.

- **Escrow Set-up:** The setup is the same as MICROPAY2 except release condition $\tilde{\Pi}_3^{\mathsf{esc}}$ is used instead of $\tilde{\Pi}_2^{\mathsf{esc}}$.

- **Payment Request:** This step is the same as in MICROPAY1; M sends the pair (c, a^M) to the payer U.

- **Payment Issuance:** This is the same as MICROPAY2. U sends M a signature σ_1 on the transaction (a^{esc}, a^M), and a signature σ on $(c, r_2, a^M, a^{\mathsf{pen}})$ (w.r.t. to $\mathsf{pk}^{\mathsf{esc}}$).

- **Claim Prize:** This is the same as MICROPAY2 except that M sends U (instead of T) the winning tuple $(x, a^{\mathsf{esc}}, a^M)$. U verifies the witness and then signs transaction (a^{esc}, a^M) using key $\mathsf{sk}_2^{\mathsf{esc}}$ and sends the resulting signature σ_2 to M. M uses signatures σ_1 and σ_2 to spend escrow a^{esc}.

  If U does not send M a valid signature σ_2 within a certain timeout, then M invokes the Resolve Aborted Prize method.

- **Resolve Aborted Prize:** When T receives a tuple $(x, a^{\mathsf{esc}}, a^M)$ such that $x = (c, r_1, s, r_2, \sigma)$, $c = \mathbf{Com}(r_1; s)$, σ is a valid signature on (c, r_2, a^M) with respect to $\mathsf{pk}_1^{\mathsf{esc}}$, and if the last 2 digits of $r_1 \oplus r_2$ are 00, T signs (a^{esc}, a^M) with respect to pk^T. M uses signatures σ and this new σ_T to spend escrow a^{esc}.

- **Penalty:** T performs the same penalty tasks as in MICROPAY2.

Security Analysis It follows using standard cryptographic techniques that the same security claims that held with respect to MICROPAY2 also hold for MICROPAY3. Additionally, note that if U and M are both executing the protocol honestly, T is never invoked.

2.4 Making Our Schemes Non-interactive

In all of our MICROPAY schemes, the merchant must send the first message to the payer, which is followed by the payer "confirming" the transaction. In some situation it may be desirable for the merchant to be able to post a *single, fixed* first message, that can be resued for an any number of users (payers) and any number of transactions (and the payer still just sends a single message confirming the transaction).

We generalize ideas from Micali and Rivest [15][4] to modify our scheme to be non-interactive in this respect. We present this technique concretely for the MICROPAY1 scheme, but note that the technique applies to all of our schemes. This technique requires each transaction to be uniquely identified by both Payer and Merchant; e.g. the rough time-of-day and IP-address of the payer and merchant, which we denote as t, can be used to identify the transaction.

[4]The central difference is that we rely on a verifiable unpredictable function (VUF), whereas [15] rely on a verfiable random function (VRF); see [14] for definitions of these notions. Relying on a VUF enables greater efficiency.

Merchant Set-up: The merchant samples a verifiable unpredictable function (VUF) [14] f_m and a bitcoin address a_M and publishes f_M, a_M.

Escrow Set-up: The payer follows the same instructions to setup an escrow; the release condition for the escrow requires a witness (σ, y, π, t, a_M) such that

1. σ is a signature on (t, a_M, f_M) with respect to $\mathsf{pk}^{\mathsf{esc}}$
2. π certifies that $f_M(\sigma) = y$ (recall that each VUF is associated with a proof systems which enables certifying the output of the VUF on a particular input).
3. $H(y)$ begins with 00, where H is a hash function modeled as a random oracle.

Payment Issuance: To send a probabilistic payment of $X/100$ for transaction t, the payer retrieves the function f_M for the merchant, computes a signature σ on t, a_M, f_M (w.r.t. to $\mathsf{pk}^{\mathsf{esc}}$) and sends σ to the merchant. The merchant verifies that the signature is valid.

Claim prize: The merchant's ticket is said to win the lottery if $H(f_m(\sigma))$ begins with 00.

Efficient instantiations of VUFs Practical VUFs in the Random Oracle Model can be based on either the RSA assumption (as in [15]), or the Computational Diffie-Hellman assumption, as we now show. This new VUF (which leads to greater efficiency than the RSA based one used in [15]) is the same as a construction from [9] but for our purposes we only need to rely on the CDH assumption (whereas [9] needs the DDH assumption). Let G be a prime order group in which the CDH problem is hard and g is a generator. The VUF is indexed by a secret seed $r \in \mathbb{Z}_q$, and the public description of the function is the tuple (G, g, g^r). On input y, the VRF evaluates to $H(y)^r$, where H is a random oracle, and produces a proof π which is a non-interactive zero-knowledge proof in the random oracle model that the pair $(g, g^r, H(y), H(y)^r)$ form a DDH triple.

3 Implementation in Bitcoin

In this section, we describe how our schemes can be implemented in Bitcoin. We begin with a more formal description of the Bitcoin protocol.

3.1 Formal description of the Bitcoin protocol

A ledger consists of an ordered sequence of blocks, each block consists of a sequence of transactions. Blocks and transactions are uniquely identified by a hash of their contents. Each transaction contains a sequence of *inputs* and a sequence of *outputs*. An input consists of a triple (t_{in}, i, ω) where t_{in} is the identifier (hash) of a previous transaction, i is an index of an output in transaction t_{in}, and ω is the *input script* or the "cryptographic witness to spend the ith output of transaction t_{in}." The i^{th} output of a transaction t consists of a triple $(a, x, \Pi_{t,i})$ where a is an address, x is an amount of Bitcoin, and $\Pi_{t,i}$ is a "release condition", i.e. a predicate that returns either true or false. A "cryptographic witness" to spend an output (t_{in}, i) is a string ω such that $\Pi_{t_{in},i}(\omega) = 1$.

An *address* a is formed from the public key of an ECDSA key pair as follows: generate an ECDSA key pair $(a_{\mathsf{sk}}, a_{\mathsf{pk}})$, then compute the hash $h \leftarrow 00||\textsc{ripemd}\text{-}160(\textsc{sha}256(a_{\mathsf{pk}}))$, compute a checksum $h' \leftarrow \textsc{sha}256(\textsc{sha}256(h))$, and finally compute the address $a \leftarrow \textsc{base}58(h||h'_{1,4})$ where $h'_{1,4}$ are the first four bytes of h' and $\textsc{base}58$ is a binary-to-text encoding scheme[5]. Thus, given a public key pk, one can verify that it corresponds to a particular address a_{pk}.

Suppose the i-th output of transaction t_{in} is (a', Π', x'). An input (t_{in}, i, ω) is valid if the following holds: (a) ω and Π' can be interpreted as a bitcoin script, and (b) after executing ω and then executing Π' on a stack machine, the machine has an empty stack and its last instruction returns true. A transaction is considered valid if each of its inputs are *valid*, and the sum of the amounts of the inputs is larger than the sum of the amounts of the outputs.

A standard release condition Π^{std} mentioned earlier in this paper requires a signature of the current transaction using a key specified in Π^{std}. This condition is specified in the Bitcoin scripting language as follows:

$$\textsc{dup} \quad \textsc{hash}160 \quad [\textsc{h}(\textsc{pk})] \quad \textsc{eq\_verify} \quad \textsc{checksig}$$

An input script that satisfies this condition is $\omega = [\sigma]\ [pk]$.

To illustrate, we briefly describe the steps to check the release condition Π^{std} with script ω. First, $\omega = [\sigma]\ [pk]$ is interpreted, which pushes the string σ and the string pk onto the stack. Next, Π^{std} is interpreted. It first duplicates the argument on the top of the stack (pk), then hashes the duplicated argument, pushes the hash of a particular public key pk onto the stack, verifies the equality of the first two arguments on the stack (which should be the string $\textsc{h}(\mathsf{pk})$ that was just pushed onto the stack and the hash of the public key given by the input script ω), and if equal, then checks the signature on the current transaction[6] using the next two arguments on the stack which are pk and σ.

Another common release condition is called a *script hash*. In this case, the release condition only specifies a hash of the release condition script. This condition is usually coded in the scripting language as

$$\textsc{hash}160 \quad [h] \quad \textsc{eq\_verify}$$

which is interpreted as a script that first hashes the top argument on the stack, pushes the string h onto the stack, and then verifies equality of the two. An

[5]Base58 uses upper- and lower- case alphabet characters and the numerals 1-9, but removes the upper-case O, upper-case I and lower-case l to eliminate ambiguities

[6]A very specific transformation is used to change the current transaction into a string upon which the signature σ is verified using public key pk.

input script that satisfies this condition might be $\omega = [a_1]\ \ [a_2]\ \ldots\ [a_n]\ [script]$, i.e. the script pushes arguments $a_1, \ldots, a_n$ onto the stack, and then pushes a string *script* onto the stack. When a certain bit is set in the output address, then the release condition first evaluates ω to setup the stack, then interprets the release condition which checks that the first argument $[script]$ on the stack is the same one specified in the release condition, and then interprets $[script]$ as a *new* script which it then executes against the values $a_1, \ldots, a_n$ which remain on the stack.

A script hash is the preferred method for encoding *multi-signature* release conditions, i.e. transactions which require more than one party to sign for the release condition to be satisfied. An example such as

$$2\ [\mathsf{pk}_1]\ [\mathsf{pk}_2]\ 2\ \textsc{check\_multisig}$$

pushes $2, \mathsf{pk}_1, \mathsf{pk}_2, 2$ onto the stack and then runs the CHECK_MULTISIG operation which then reads these 4 arguments and "succeeds if the next two arguments on the stack correspond to signatures under 2 of the public keys $\mathsf{pk}_1, \mathsf{pk}_2$." To satisfy this script, the witness should be of the form $\omega = 0\ \sigma_1\ \sigma_2$ where σ_i is a signature on the transaction under key sk_i. The extra 0 at the beginning is a peculiarity of this operation's syntax.

3.2 Modifications to Bitcoin for MICROPAY1

The Bitcoin script language supports a CHECK_SIG operation that reads a public key and a signature from the stack and then verifies the signature against the public key on a message that is derived in a special way from the current transaction. This (and its multi-sig version) is the only operation that performs signature verification. In MICROPAY1, however, our scheme requires the verification of a signature on a transcript of a coin-tossing protocol, i.e. step (3) of the release condition $\Pi^{\mathsf{esc}}(x, a_{\mathsf{esc}}, a_2)$ needs to verify a signature on the tuple (c, a_2, r_2). Thus, to support our protocol, we suggest a new operation CHECK_RAWSIG which reads a public key, a signature, *and* values from the stack which it concatenates to produce the message that is used to check the signature. More specifically, when this instruction is called, the top of the stack should appear as follows:

$$[a_n]\ \cdots\ [a_1]\ [n]\ [\sigma]\ [\mathsf{pk}]$$

The operation performs the following steps:

1. Read the top argument on the stack; interpret as a public key. (Same as the first step of CHECK_SIG.)
2. Read the next argument on the stack; interpret as a signature string. (Same as the second step of CHECK_SIG.)
3. Read the next argument n from the stack and interpret as a 32-bit unsigned integer.

4. Read the next n arguments $a_n, a_{n-1}, \ldots, a_1$ from the top of the stack and concatenate to the string $m = a_1||a_2||\cdots||a_n$ where $||$ is a unique delimiter string.

5. Verify that σ is a signature on message m under public key pk. If not, then abort.

Thus, the only difference between this instruction and the OP_CHECKSIG instruction is how the message m is constructed. In the later case, the message is constructed by removing part of the script from the current transaction in a specific way. An implementation of this method in the `libbitcoin`[8] library requires only 30 additional lines of code.

Additionally, in order to verify that the transcript of our "coin-flipping" protocol is a winning transcript, we need to add (or xor) integers on the top of the stack and compare and integer on the stack to a fixed constant. In the current scripting language, numeric opcodes such as ADD and LTCMP are restricted to operating on 4-byte integers. To ensure the soundness of our coin-flipping protocol, however, we require the merchant to select a witness x (that is used to form the commitment c) from 128-bit strings. Thus, the integers on our stack will be larger than 4-bytes, and currently, the Bitcoin script stops evaluating the script and fails when this event occurs. To enable our functionality, we require the operations to *truncate* the integers on the stack to 4-byte values and continue evaluating the script (instead of aborting the execution of the script as they do now). This change requires only five lines of code in `libbitcoin`.

Finally, in order to implement the penalty condition, the script must verify the existence of signatures on distinct winning lottery tickets. These operations can be performed using the same two above changes.

3.3 Implementing MICROPAY2

We implement our second scheme in this section. Figure 2 shows the message flow; we then describe each message in detail.

STEP 0. The VTS T publishes a public key pk^T and retains a secret key sk^T used for signing.

EXAMPLE: Party T publishes public key

```
0305a8643a73ecddc682adb2f9345817d
c2502079d3ba37be1608170540a0d64e7
```

STEP 1. The first step of our scheme is for the user to post an *escrow* and a *penalty escrow* transaction for $100X$ and λX respectively onto the blockchain. To do so, the payer generates new addresses a^{esc} and a^{pen} while retaining the associated key pairs $(\mathsf{sk}^{\mathsf{esc}}, \mathsf{pk}^{\mathsf{esc}}, \mathsf{sk}^{\mathsf{pen}}, \mathsf{pk}^{\mathsf{pen}})$, and publishes a transaction on the ledger that specifies outputs a^{esc} and a^{pen} with the escrow release output script[7].

[7] Bitcoin convention suggests the use of a scripthash to implement a multi-signature transaction, but in this case, the explicit script allows public verification of an escrow.

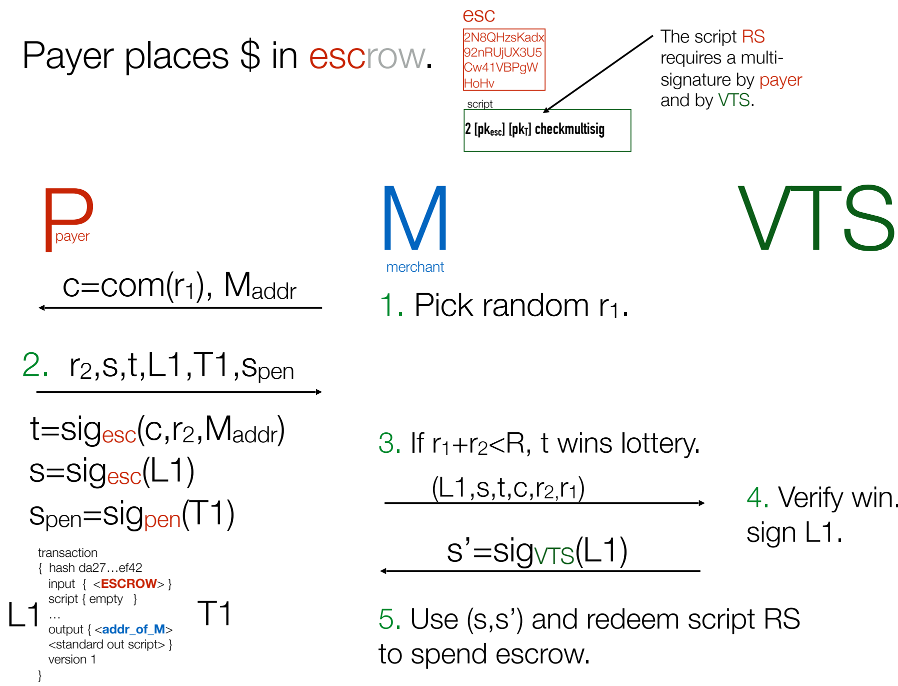

Figure 2: An example of how MICROPAY2 can be implemented in the Bitcoin scripting language.

The output script for esc will be

$$“2 \ \ [\mathsf{pk}^{T}] \ \ [\mathsf{pk}^{\mathsf{esc}}] \ \ 2 \ \ \text{CHECK\_MULTISIG}”$$

The script for pen is constructed similarly.

EXAMPLE: The user creates a transaction to post the escrow to the blockchain:

```
transaction
{
    hash e164...099d
    inputs
    {
        input
        {
            previous_output
            {
                hash da27...6979
                index 0
            }
            script ""
```

```
            sequence 4294967295
        }
    }
    lock_time 0
    outputs
    {
        output
        {
            script "2 [ 0405...005d ] [ 0469...fe2f ] 2 checkmultisig"
            value 1000000
        }
        output
        {
            script "2 [ 04e1...e59b ] [ 0469...fe2f ] 2 checkmultisig"
            value 10000000
        }
    }
    version 1
}
```

The escrow address for this example is `mgac...RPUC` (derived from the first public key in the first output) and the penalty address is `mihF...3KH4`. Before posting this transaction to the blockchain, the User and T first exchange signatures on transactions which allow U to recover the penalty escrow after a given time-out. Specifically, using the transaction hash `e164...099d`, U creates a transaction t_1 that spends output index 1 to an address that U creates with a locktime that reflects roughly 6 months in the future, and sends t_1 to T. Next, T signs t_1 using pk_T and sends it back to U. (These steps have been omitted in the above.) Finally, U posts transaction `e164...099d` to the blockchain. As soon as it is confirmed, U can begin spending.

STEP 2. To request a payment, the merchant picks a random $r_1 \leftarrow \{0,1\}^{128}$ string and then computes $c \leftarrow H(r_1)$ where H is the SHA256 operation implemented in the Bitcoin scripting language. The merchant also generates a new Bitcoin address a_2 and sends (c, a_2) to the payer while retaining the public and secret keys associated with a_2.

EXAMPLE The Merchant picks the random message

$$r_1 \leftarrow \texttt{29c14f18638da11b75663e050087b591}$$

computes $c \leftarrow \textsc{sha}256(r_1)$ and a new bitcoin address

$$c = \begin{array}{l}\texttt{7c12e848a4a3a9f31c7abea5ab323eeb}\\ \texttt{6893c3a08675cc6c076e39950e52695e}\end{array}$$

$$a_2 \leftarrow \texttt{mkKKRLweRbu7Dam82KiugaA9bcnYXSyAVP}$$

and sends the message (c, a_2).

STEP 3. Upon receiving (c, a_2) from a merchant, the payer verifies that c is the proper length for the hash of a 128-bit string, and that a_2 is a well-formed bitcoin address. The payer picks a random 8-bit string $r_2 \leftarrow \{0,1\}^8$, and then uses $\mathsf{sk}^{\mathsf{esc}}$ in order to compute the signature σ on the message (c, r_2, a_2) using the secret key $\mathsf{sk}^{\mathsf{esc}}$. The payer also computes a signature σ_1 on the transaction (a^{esc}, a_2) using the secret key $\mathsf{sk}^{\mathsf{esc}}$ and a signature σ_{pen} on a transaction t_{pen} that spends the penalty escrow to an invalid address like. The payer sends $(a^{\mathsf{esc}}, r_2, \sigma, \sigma_1, \sigma_{\mathsf{pen}}, t_{\mathsf{pen}})$ to the merchant.

EXAMPLE The payer randomly samples $r_2 \leftarrow \mathsf{37}$ and then computes a signature on (c, r_2, a_2) as

$$\sigma \leftarrow \texttt{IKZRV...rgXLHs=}$$

The payer then forms the transaction (a^{esc}, a_2) as follows

```
transaction {
 hash 2de3...0e73
 inputs {
  input {
   previous_output {
                hash fc72...d347
                index 0
   }
   script ""
   sequence 4294967295
  }
 }
 lock_time 0
 outputs {
  output {
   address mkKK...yAVP
   script "dup hash160 [ 34a...e2a ] eq_ver chksig"
   value 100000
  }
 }
 version 1
}
```

and then signs the transaction using $\mathsf{sk}^{\mathsf{esc}}$

$$\sigma_1 \leftarrow \texttt{3044...ed01}$$

STEP 4. Upon receiving $(r_2, \sigma, \sigma_1, \sigma_{\mathsf{pen}}, t_{\mathsf{pen}})$ from the payer, the Merchant first verifies the two signatures on the respective messages and verifies that a^{esc} has not yet been spent. The merchant then checks whether $r_1 \oplus r_2$ results in a string whose last two (or alternatively, first two) digits are zero.

If so, then the merchant has a winning ticket. To redeem the escrow amount, the merchant sends the winning tuple consisting of $x = (c, r_1, r_2, \sigma, \sigma_1, a^{\mathsf{esc}}, a_2)$

to the VTS T. T verifies that the tuple corresponds to a win for the escrow a^{esc}, and if so, then signs the transaction (a^{esc}, a_2) using public key pk^T. Specifically, T verifies that $c = H(r_1)$, σ is a valid signature on (c, r_2, a_2) w.r.t. $\mathsf{pk}^{\mathsf{esc}}$, and the last 2 digits of $r_1 \oplus r_2$ are 00.

Furthermore, T publishes tuple x on its own bulletin board, on the bitcoin blockchain, or on some "alt-chain."

If a^{esc} has already been spent when T receives the winning tuple x, then T looks up the winning tuple x' used to claim a^{esc}. T then has evidence that U has double-spent the escrow. T then finds transaction t_1 and signature σ_t, signs t_1 itself using sk_T to produce σ'_t, and then spends a^{pen} to address 0 by sending $(t_1, \sigma_t, \sigma'_t)$ to the cryptocurrency network.

STEP 5. Finally, once M has received the signature σ_T from T, then M can spend a^{esc} to address a_2 (which it controls) using σ_1, σ_T to satisfy the release condition.

4 Evaluation

4.1 Expected Revenue and Expenditure

With each of our probabilistic payment schemes, the seller receives X coins in *expectation* for every interaction with a buyer. We provide a statistical analysis to guarantee that after sufficiently many payments, both the buyer and the seller respectively spend and receive an amount that is *close* to the expectation with high probability.

Our scheme is parameterized by ρ, the probability that a lottery ticket wins. One can tune ρ to balance the number of winning transactions with the variance in the actual cost/revenue from each transaction. Although the previous section used $\rho = \frac{1}{100}$, our implementation uses ρ that is a power of 2 to simplify the coin-flipping protocol. Thus, in the following sections, we consider $\rho_1 = \frac{1}{128}$ and $\rho_2 = \frac{1}{512}$. A standard Bernoulli analysis suffices because the security properties of our scheme prevent even malicious parties from biasing independent executions of the protocol. Let R_i be a random variable denoting revenue from the i^{th} execution of the protocol (e.g., R_i is either 0 or X/ρ, in our case, either 0 or 128). The total revenue is therefore $R = \sum_i^n R_i$. As discussed previously $E[R_i] = \rho \cdot X/\rho = X$, so $E[R] = Xn$. Recall that the probability that revenue is exactly Xk is

$$\Pr[R = Xk] = \binom{n}{k\rho} (\rho)^{k\rho} (1-\rho)^{n-k\rho}$$

Using this formula and $\rho_1 = \frac{1}{128}$, we illustrate the probability of paying (or receiving) a specific amount per transaction in Fig. 3. These graphs show that both the buyer (who may, say, make 1000 transactions per year) and a seller (who may receive 10,000 transactions per month), the average price over those transactions will be close to the expected amount of X per transaction. The blue sections of those graphs show 99% of the probability mass.

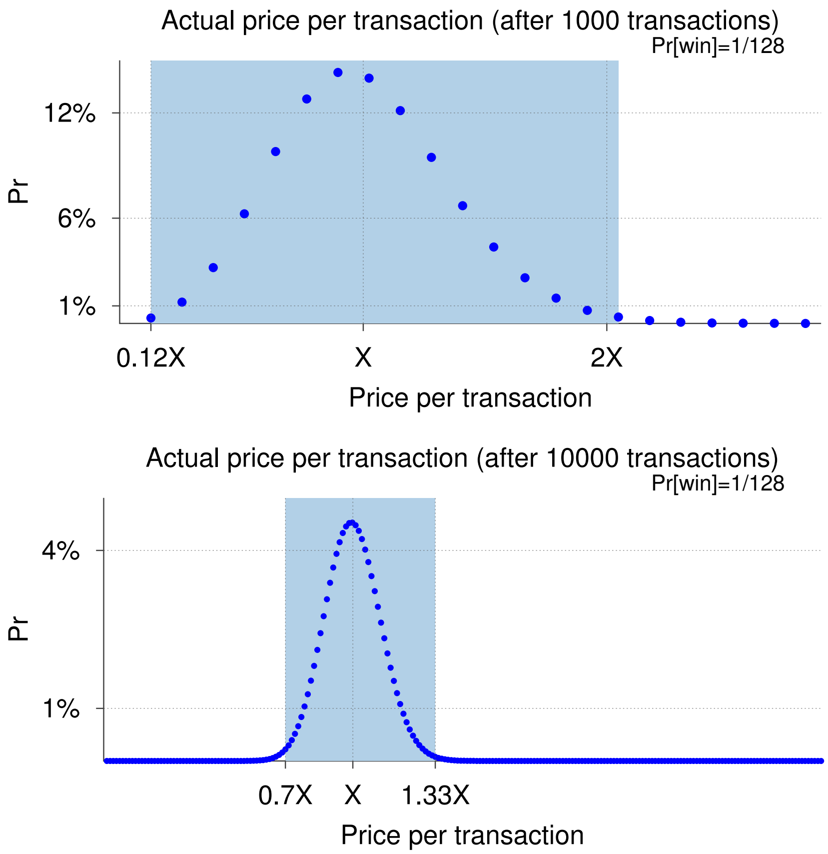

Figure 3: Pr of payment amount (parameterized by X) after 1,000 and 10,000 transactions (for win rate $\rho_1 = \frac{1}{128}$). The blue region shows 99% of the mass. If escrow is $128X$, then the expected payment is X.

As the number of transactions increases for a very busy seller (e.g, a web site that receives millions of views), the guarantees on revenue become even tighter. To illustrate, we now compute the probability that $R < 0.8n$, i.e., that revenue is less than 80% of the expected value:

$$\Pr[R < 0.8n] = \sum_{k=0}^{\lfloor 0.8n\rho \rfloor} \Pr[R = \rho \cdot k]$$

The floor function in the summation's upper bound make the function "choppy" and non-monotone at those n when the value discretely increases by 1. The Chernoff bound is a general tool that can be used to bound tail inequalities such as this one. However, this estimate is loose, and we instead compute the exact value in Fig. 4. After 100,000 transactions, there is high probability that the actual revenue will be at least 80% of the expected value and good probability that the revenue will be at least 90% of the expected. In Fig. 5, we show the same results for win rate $\rho_2 = \frac{1}{512}$.

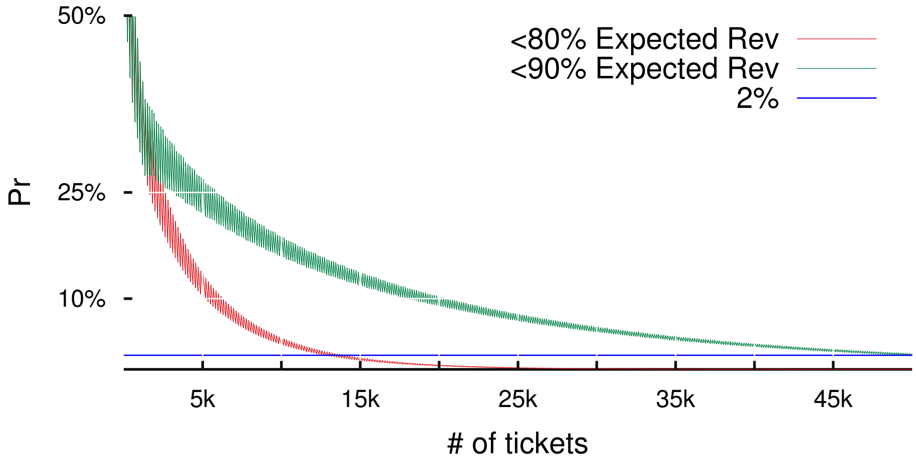

Figure 4: For win rate $\rho_1 = \frac{1}{128}$, probability that the seller's revenue is less than 80% and 90% of the expected revenue. The curves have a "sawtooth" pattern due to discreteness. At 15,000 and 50,000 transactions, there is a roughly 2% chance that revenue is less than 80% or 90% respectively of the expected revenue.

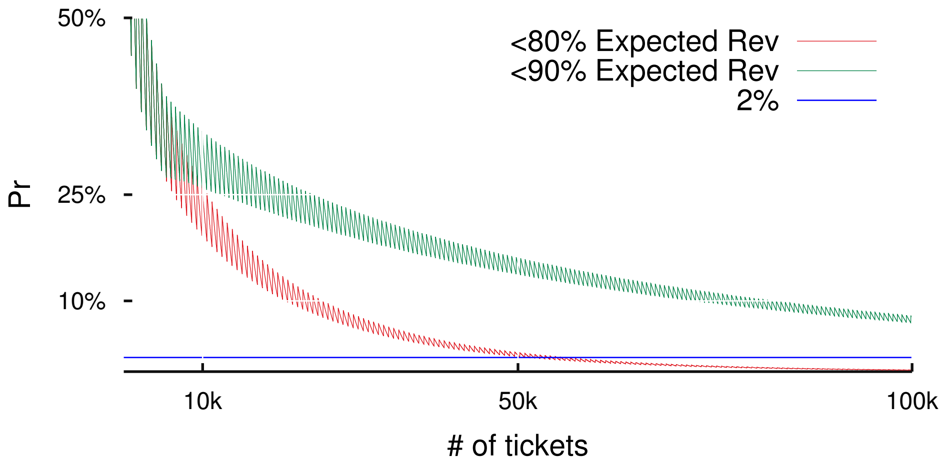

Figure 5: For win rate $\rho_2 = \frac{1}{512}$, probability that the seller's revenue is less than 80% and 90% of the expected revenue.

4.2 Performance of the Scheme

Our schemes are all highly efficient; the first message from the seller requires only a hash computation (and optionally the creation of a new address, in the fast version of this step, we reuse the same bitcoin address for all merchant transactions[8]). The second message from the buyer requires the computation of two signatures. The final check to determine whether the transaction is paying requires two signature verifications and one comparison operation. We first show micro-benchmarks for each of these operations, and then demonstrate how the scheme operates in a real system.

Micro-benchmarks for each operation

OPERATION	AVG TIME (μs)	95% CI (μs)
Request Ticket	84.9	± 2.56
Request Ticket (Fast)	3.7	± 0.12
Make a Ticket	170.6	± 5.28
Check Ticket	437.6	± 10.45
VTS Check	496.1	± 6.60

These measurements where taken on an Intel Core i7-4558U CPU @ 2.80GHz, with 2 cores, 256 KB of L2 cache per core, 4MB of L3 cache, and 16GB of RAM. Each function was profiled using the Go language benchmark framework which called the function at least 10000 times to time the number of nanoseconds per operation. The Go benchmark framework was run 50 times and averaged to report the sample time and the 95% confidence interval reported in the table. Only one core was used during the testing. As the table demonstrates, the protocol messages can be generated in microseconds, with ticket checking requires less than half a milli-second. Thus, the overhead of the protocol is very low in terms of computation.

In terms of communication, we have made no effort to compress or minimize the size of the messages. For ease of implementation, we use base64 encodings for the signatures, commitments, and addresses in the protocol (rather than a more efficient binary encoding). In the table below, we report the size (in bytes) for each of the messages. The ticket message has a variable size because it includes two signatures whose message sizes are variable.

[8] Although Bitcoin specifications suggest that each transaction use a totally new address, with proper key management on behalf of the merchant, there is no reason the same address cannot be used to receive many payments.

OPERATION	MESSAGE SIZE (bytes)
Request Ticket	73
Request Ticket (Fast)	73
Make a Ticket	398 ± 10
Check Ticket	-
VTS Check	-

4.3 Experiments in a sample web server

To illustrate how our scheme can be used to sell "content" on the Internet, we developed a webserver that serves dynamic pages and also implements our MICROPAY2 protocol. Our experiment shows that the overhead of adding the messages of the micropayment protocol add little in terms of performance penalty to the infrastructure. The most expensive operation on the server is to verify the lottery ticket (i.e., check two signatures), and this adds less than half a milli-second to the server response time—a value that is essentially masked by the variance in network performance.

In practice, we envision our system as a proxy that sits in front of a legacy content server and only handles the micropayment; this experiment serves as an illustrative benchmark for that architecture. In particular, it shows that a basic and unoptimized server can handle millions of tickets.

Design We implemented a webserver using the Go `net/http` package. The server handles three kinds of requests, `\base`, `\ask`, and `\buy`. The `\base` endpoint returns a page that is rendered with a template and a dynamic parameter (to model the fact that it is not simply a static page that is cached in memory). The size of this page is roughly 2kb. This endpoint serves as a control for our experiment to understand the baseline performance of our webserver implementation. Next, the `\ask` endpoint returns the first message of our micropayment scheme, i.e. a request for a ticket. This method models what a buyer's client queries in order to receive a ticket request[9]. Finally, the `\buy` endpoint accepts the second message (the ticket) of our micropayment protocol and checks whether the ticket is well-formed and whether the ticket is a winning one. If the ticket is well-formed, the method returns the same dynamically generated webpage as the `\base` method. Thus, the combination of making an `\ask` query and then a `\buy` query reflects the overhead of processing a micropayment before serving content.

Compute-bound experiment In the first experiment, we measured the extent to which the extra computation for a server would become a bottleneck at

[9]In practice, the first message will be embedded in the link to the content that requires a payment, hence the most expensive component of this message—the network cost— can essentially be hidden from the user's experience.

Internet scale. We ran a client that made both *control* and *experiment* requests from a 2-core/4-hyperthread laptop running on a university network from the east coast. The control experiment makes a call to `\ask` and then `\base`; the experiment makes a call to `\ask` and `\buy`. Our experiment attempts to isolate the difference between calling just `\base` and accessing the same content through `\buy`; but in order to perform the latter, we need to have information conveyed through `\ask`. This extra round-trip is hidden in practice because it is bundled with the (several) calls to a server that are used to access the "homepage" from which the links to content-for-purchase are conveyed. Thus, to avoid comparing one round-trip against two, both of the experiments make a call to `\ask`.

The client issued 25000 requests using 20 threads for at least 20 seconds; each thread pooled its network connection to amortize network overhead over the requests. Each run (to either control or experiment) was performed 30 times over the course of a day and a delay of at least 15 seconds was introduced between runs to allow network connections to gracefully terminate. The client sent its queries from the east coast. The server used a single core on a `t4.xlarge` instance from the US-East region of EC2 which has an Intel Xeon CPU E5-2666 v3 @ 2.90GHz and 8GB of memory.

As illustrated by the table below, the difference between the performance of the `\base` system and `\buy` are overwhelmed by network timing noise; the confidence interval of the experiment roughly matches the microbenchmark timings for the `\buy` calls.

OPERATION	REQ/SEC	AVG RESP TIME (95% CONF INT)
`\base`	534	1.87 ± 0.26 ms
`\buy`	497	2.01 ± 0.30 ms

Extrapolation When run as a proxy, a micropayment server with 8 cores/16 threads can handle at least 4000 transactions per second, or roughly 350 million page views per day. At roughly 600 bytes per message to account for protocol overheads, this amounts to a bandwidth overhead for micropayment messages of merely $600 * 4000 = 2.4$mb/sec.

Network Test The previous tests did not include network connection overhead. We ran the same experiment using a single thread making a single request, serially, 2000 times with a 2 second delay between each request. The client ran from a laptop connected to the internet over a cable modem. Figure 6 plots a histogram of the response times for the control and experiment. The two distributions are very close as expected.

VTS Performance In MICROPAY2, the VTS signs all winning lottery transactions. At Internet scale, this party could become a bottleneck since every winning ticket must be processed in near real-time to mitigate double-spending attacks. Based on the microbenchmarks in the previous section, a single core can

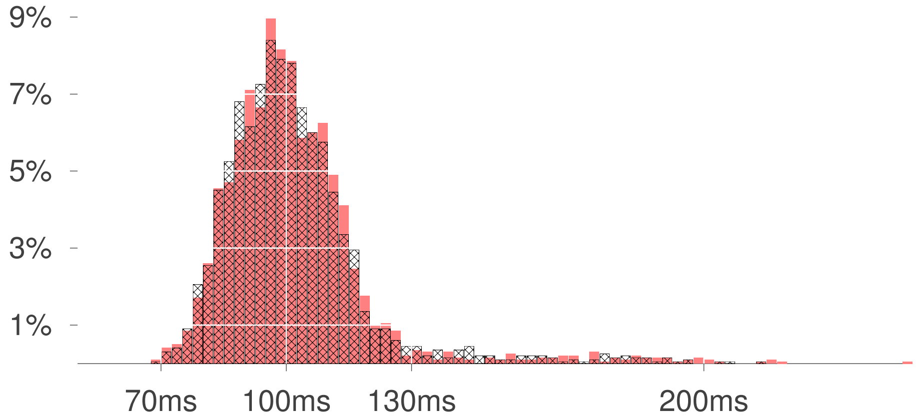

Figure 6: A histogram of response times for a single request over a cable modem. The base red is the experiment, the overlaid checkerbox is the control.

also verify and sign 2000 winning tickets per second. Including networking overhead extrapolated from our first experiment, we estimate that a micropayment server with 8 cores/16 threads can handle at least 4000 *winning* transactions per second, or roughly 350 million winning lottery tickets per day. When the winning ratio parameter is $\rho_1 = \frac{1}{128}$, roughly 1 out of 128 tickets will be winning, and thus, a single VTS server can theoretically support 512,000 *global* micropayment content views per *second*, or $\sim$44 billion total micropayment content views per day. The later number assumed uniform access rate throughout the day, but real traffic follows cyclic patterns with peak times that are much busier than off-peak times. These are theoretical maximums, but after adding redundancy for robustness, this analysis and experiment suggests that a small set of servers suffice to handle Internet scale transaction processing.

Another potential bottleneck occurs with the underlying cryptocurrency bandwidth. As the graph in Fig. 7 depicts, during 2015, the number of daily Bitcoin transactions processed on the blockchain hovers around 10^5. The current Bitcoin protocol can only handle 7 transactions per second on average, or roughly 10^6 transactions per day, and thus, at parameter ρ_1, it seems feasible for the current Bitcoin protocol to handle roughly 10^8 total paid transactions. Many research efforts are underway to increase the bandwidth for the number of transactions by a factor of 10x to 100X, and our scheme's scalability naturally benefits from these advancements. We can also decrease the ρ_1 value to improve the scalability (at the cost of increasing the variance of expected revenue and costs for the sellers and buyers).

Acknowledgements

We would like the thank the anonymous reviewers for helpful comments on this draft. We also want to thank Joe Bonneau for helpful discussions and in

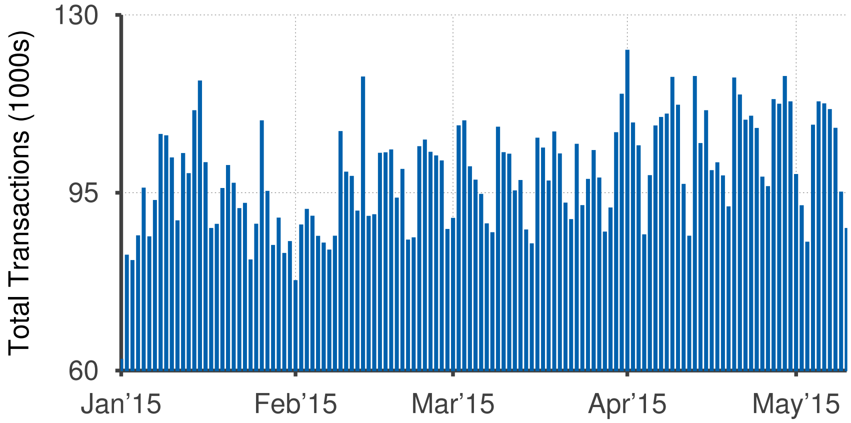

Figure 7: Bitcoin transactions per day (2015)

particular, pointing out issues with front-running in an earlier version of our protocols.

References

[1] Ross Anderson, Charalampos Manifavas, and Chris Sutherland. Netcard—a practical electronic-cash system. In *Security Protocols*, pages 49–57, 1997.

[2] Marcin Andrychowicz, Stefan Dziembowski, Daniel Malinowski, and Lukasz Mazurek. Secure multiparty computations on bitcoin. In *IEEE S&P*, 2014.

[3] Iddo Bentov and Ranjit Kumeresan. How to use bitcoin to design fair protocols. In *CRYPTO'14*, 2014.

[4] Jon Bonneau, Arvind Narayanan, Andrew Miller, Jeremy Clark, Joshua A. Kroll, and Edward W. Felten. Mixcoin: Anonymity for bitcoin with accountable mixes. `eprint.org/2014/77`, 2014.

[5] David Chaum. Achieving electronic privacy. *Scientific American*, pages 96–101, August 1992.

[6] David Chaum, A. Fiat, and M. Naor. Untraceable electronic cash. In *CRYPTO'89*, 1989.

[7] Etherium Foundation. White paper. `https://github.com/ethereum/wiki/wiki/White-Paper`, 2015.

[8] Libbitcoin foundation. A bitcoin library. `https://github.com/libbitcoin/libbitcoin/`, 2015.

[9] Matthew Franklin and Haibin Zhang. Unique ring signatures: A practical construction. In *Financial Cryptography and Data Security*. 2013.

[10] Steve Glassman, Mark Manasse, Martin Abadi, Paul Gauthier, and Patrick Sobalvarro. The millicent protocol for inexpensive electronic commerce. In *WWW'95*, 1995.

[11] Oded Goldreich. *Foundations of Cryptography — Basic Tools*. Cambridge University Press, 2001.

[12] Richard J. Lipton and Rafail Ostrovsky. Micro-payments via efficient coin-flipping. In *Financial Cryptography*, pages 1–15, 1998.

[13] G. Medvinsky and C. Neuman. Netcash: A design for practical electronic currency on the internet. In *CCS'94*, 1994.

[14] Silvio Micali, Michael Rabin, and Salil Vadhan. Verifiable random functions. In *FOCS*. IEEE, 1999.

[15] Silvio Micali and Ronald L. Rivest. Micropayments revisited. In *CT-RSA*, 2002.

[16] Board of Governors of the Federal Reserve System. Federal reserve board press release 29 june 2011. `http://www.federalreserve.gov/newsevents/press/bcreg/20110629a.htm`, June 2011.

[17] Joseph Poon and Thaddeus Dryja. The bitcoin lightning network: Scalable off-chain instant payments, v.0.5.9.2. `https://lightning.network/lightning-network-paper.pdf`, 2016.

[18] Ron Rivest and Adi Shamir. Payword and micromint: Two simple micropayment schemes. In *Cambridge Workshop on Security Protocols*, 1996.

[19] Ronald L. Rivest. Electronic lottery tickets as micropayments. In *In Financial Cryptography*, 1997.

[20] David Wheeler. Transactions using bets, 1996.

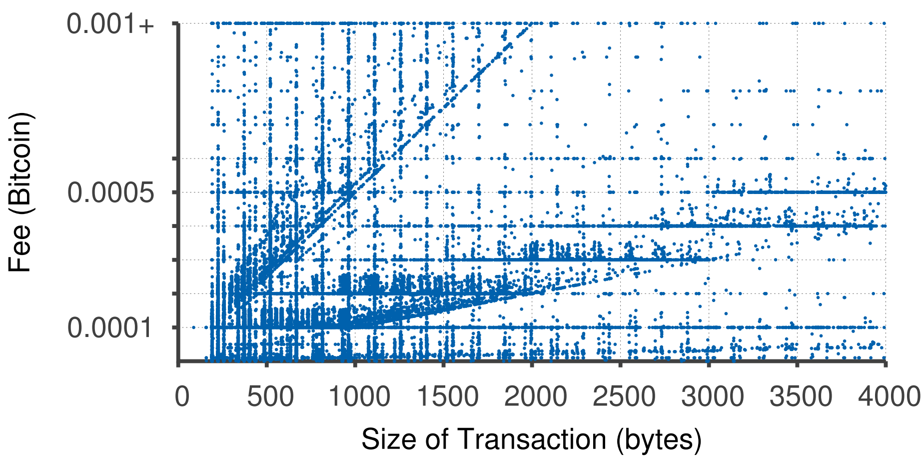

Figure 8: A plot of transaction fee versus transaction size for one million Bitcoin transactions that occurred in May 2015. The Bitcoin specification suggests that each transaction should pay roughly 0.0001 bitcoin per kilobyte (rounded up) of transaction data.