Decentralized Anonymous Micropayments

Alessandro Chiesa
alexch@berkeley.edu
UC Berkeley

Matthew Green
mgreen@cs.jhu.edu
Johns Hopkins University

Jingcheng Liu
liuexp@berkeley.edu
UC Berkeley

Peihan Miao
peihan@berkeley.edu
UC Berkeley

Ian Miers
imiers@cs.jhu.edu
Johns Hopkins University

Pratyush Mishra
pratyush@berkeley.edu
UC Berkeley

October 31, 2016

Abstract

Micropayments (payments worth a few pennies) have numerous potential applications. A challenge in achieving them is that payment networks charge fees that are high compared to "micro" sums of money.

Wheeler (1996) and Rivest (1997) proposed probabilistic payments as a technique to achieve micropayments: a merchant receives a macro-value payment with a given probability so that, in expectation, he receives a micro-value payment. Despite much research and trial deployment, micropayment schemes have not seen adoption, partly because a trusted party is required to process payments and resolve disputes.

The widespread adoption of decentralized currencies such as Bitcoin (2009) suggests that decentralized micropayment schemes are easier to deploy. Pass and Shelat (2015) proposed several micropayment schemes for Bitcoin, but their schemes provide no more privacy guarantees than Bitcoin itself, whose transactions are recorded in plaintext in a public ledger.

We formulate and construct *decentralized anonymous micropayment* (DAM) schemes, which enable parties with access to a ledger to conduct offline probabilistic payments with one another, directly and privately. Our techniques extend those of Zerocash (2014) with a new privacy-preserving probabilistic payment protocol. One of the key ingredients of our construction is *fractional message transfer* (FMT), a primitive that enables probabilistic message transmission between two parties, and for which we give an efficient instantiation.

Double spending in our setting cannot be prevented. Our second contribution is an economic analysis that bounds the additional utility gain of any cheating strategy, and applies to virtually any probabilistic payment scheme with offline validation. In our construction, this bound allows us to deter double spending by way of advance deposits that are revoked when cheating is detected.

Keywords: decentralized currencies; micropayments; probabilistic payments; anonymity; Bitcoin

Contents

1 Introduction

We formulate and construct *decentralized anonymous micropayments*, by way of probabilistic payments.

Micropayments. A *micropayment* is a payment of a small amount, e.g., a fraction of a penny [Whe96, Riv97]. Micropayments have many potential applications, including advertisement-free content delivery, spam protection, rewarding nodes of P2P networks, and others. Achieving micropayments involves at least two main challenges. First, payment processing fees dwarf "micro" payment values. Second, micropayment applications often require *fast merchant responses*, which, in many settings, are achieved via *offline payments*, which are vulnerable to double spending.

Probabilistic payments. A technique to reduce processing fees is to amortize them over multiple payments by way of *probabilistic payments* [Whe96, Riv97].[1] These are protocols that enable a customer to pay V units of currency to a merchant with probability p: with probability $1 - p$ the merchant receives a *nullpayment* that is not processed, and with probability p the merchant receives a *macropayment* that is processed. In expectation, the merchant receives pV units per micropayment, but the overhead and processing fees of these "lottery tickets" is p times smaller as only the infrequently generated macropayments are actually handled by the payment network. Constructing probabilistic payments is an area of ongoing interest in cryptography.

Centralized vs. decentralized systems. Despite extensive research and trial deployments [Whe96, Riv97, LO98, MR02, Riv04, Mic14], micropayment schemes have not seen widespread usage. This is perhaps due to them being *centralized systems*: a trusted third party is tasked with processing payments and punishing cheaters. Appointing such a party raises deployment costs, requires establishing complex business relationships between all involved (the trusted party, merchants, and customers), and makes participation conditional on certain requirements being met [vOR$^+$03].

Recent work in digital currencies has focused on *decentralized systems*, as the cost of entry and deployment appears to be lower. The most notable such currency is Bitcoin [Nak09], a widely adopted peer-to-peer payment system. Unlike traditional banking and e-cash schemes [Cha82, CHL05, ST99] where transactions are processed by a trusted party, Bitcoin utilizes a distributed public ledger known as the *blockchain* to store all transactions; these transactions are verified by network nodes in a peer-to-peer fashion.

Decentralized systems are thus potentially attractive for micropayments, because the overhead involving trusted parties is no longer a factor. However, Bitcoin processing fees are still relatively high (as of May 2016 the fee for a 1kB-transaction is $\approx \$0.20$), with present fees believed to be well below the cost of performing a transaction on the Bitcoin network [MB15]. Thus, fee amortization is still necessary. Caldwell [Cal12] first sketched probabilistic payments for Bitcoin. Recently, Pass and Shelat [PS15, PS16] also proposed three probabilistic payment schemes for Bitcoin, where, informally, the customer first puts V bitcoins in escrow, and then the customer and merchant engage in a coin-flipping protocol that allows the merchant to retrieve the escrow with probability p. Their three schemes differ in how payments are processed and how disputes are resolved.

Our privacy goal and limitations of prior work. We study the question of how to construct *decentralized anonymous micropayments* via the technique of (offline) probabilistic payments. The aforementioned prior work [PS15, PS16] provides *no more privacy than the underlying Bitcoin protocol*. And Bitcoin itself provides little to no privacy because every transaction is publicly broadcast and contains a payment's origin, destination, and amount; a user's payment history is thus readily available to any passive observer who can can link pseudonyms together or to real world identities.[2] This lack of privacy is particularly dangerous for micropayment applications because they typically involve high-volume pattern-rich payments (e.g., per-click payments while surfing the web), and sometimes necessitate user anonymity (e.g., bandwidth payments for Tor relays [BP15]).

Privacy is not merely an issue of individual users: if each coin's history is public, a customer may not be able to spend a coin at its 'declared' value due to its past. For example, a merchant may not accept coins whose past owners include certain political organizations. Privacy thus ensures a fundamental property of the currency: *fungibility*, which means that any two sets of coins with the same 'declared' total value are interchangeable, regardless of their provenance.

Prior work on privacy-preserving analogues of Bitcoin [MGGR13, DFKP13, BCG$^+$14] does not achieve probabilistic payments, and merely "plugging" these schemes into [PS15, PS16]'s approach results in subtle problems. Consider the following natural modification to Pass and Shelat's coin-flipping protocol: instead of a Bitcoin transaction, the

[1] Another technique is micropayment channels, which we discuss in Section 1.2.

[2] This is not merely a theoretical concern: extracting information from Bitcoin transactions is the subject of applied research [RH11, BBSU12, RS13, MPJ$^+$13] and commercial ventures [Ell13, Blo14, Cha15].

sender probabilistically transmits to the merchant a Zerocash transaction [BCG+14]. Despite the strong anonymity guarantees provided by Zerocash, merchants *still* learn information about their customers' spending habits, because each Zerocash transaction includes a unique serial number corresponding to the spent "coin". Since the customer sends to the merchant information about the escrow, this serial number is revealed in each micropayment. Since the same escrow is used across multiple probabilistic payments (to amortize fees), privacy of the customer is compromised because the merchant learns (1) which (macro or null) payments to him were made with the same escrow; and (2) which macropayments to other merchants were made with an escrow used for payments to him. This breach of privacy worsens if merchants share information with one another. In sum, while the above natural approach achieves "macropayment unlinkability", *micropayments are still linkable*, and thus customers have little privacy.

Double spending in offline probabilistic payments. Micropayment applications often require fast responses. In many settings, these in turn require *offline* validation: a merchant responds to a payment after only a local "offline" check, because he cannot wait for the payment network to validate the payment (this validation instead completes after the merchant's response). For example, validation takes a few minutes in Bitcoin, while responding to unconfirmed `zero-conf` transactions takes only a few seconds. We thus focus on *offline probabilistic payments*.

However, such payments are *vulnerable to double spending*, as we now explain. First, double spending *cannot be prevented* for offline payments, because, to prevent it, a merchant would have to refrain from responding to any payment before all payments up to, and including, this payment have been validated. One fallback is to detect and punish all double-spending customers. However, for offline *probabilistic* payments, not all double spending can even be detected.

Indeed, there are two types of double spending when using the same lottery ticket in two probabilistic payments: (1) both payments result in macropayments; or (2) the first payment results in a macropayment (thereby 'consuming' the ticket) while the second payment results in a nullpayment. While detecting the first type is easy, detecting the second type requires the payment network to 'know' the temporal order of all payments, because whether the nullpayment or the macropayment occurred first determines whether the two payments correspond to honest behavior (nullpayment first) or not (macropayment first). But knowing the global order of all payments (with high precision) is a strong synchronization property that is unrealistic in many decentralized settings, including that of Bitcoin, because information does not instantly reach everyone in the network.

Given that not all double spending can be detected, the "detect-and-punish" approach is effective only if the disadvantages of being punished (upon detection) outweigh the advantages of double spending. This may be plausible in the centralized setting, where customers have registered with a trusted party that can permanently ban and legally prosecute them. In the decentralized setting, however, banning has few consequences, if any: anyone can abandon old identities and use fresh new identities in their place.

Ruffing, Kate, and Schröder [RKS15] introduce "accountable assertions", which enable timelocked deposits in Bitcoin that are revoked upon evidence of double spending. Pass and Shelat [PS16] also suggest a Bitcoin-specific penalty mechanism to deter rational customers and merchants from cheating.[3] Unfortunately, both of these works do not provide an economic analysis to indicate how large a penalty should be to deter double spending. Such an analysis is crucial: how could detect-and-punish be a deterrent if double spending were to yield *unbounded* additional utility?

1.1 Our contributions

We overcome the aforementioned limitations via a combination of cryptographic and economic techniques. We adopt a "detect-and-punish" approach in which cryptography is used to retroactively detect and economically punish double spending and, separately, an economic analysis clarifies how much to punish so as to deter double spending in the first place. More precisely, we present the following three contributions.

1.1.1 Economic analysis of double spending for offline probabilistic payments

We characterize the additional utility that can be gained by double spending via offline probabilistic payments. We suppose that: (i) every probabilistic payment is backed by an advance deposit;[4] (ii) all macropayment double spends

[3] We also note that two of the three schemes in [PS15] do not support offline payments, and the remaining one only provides "fast online payments" where an online (publicly verifiable) trusted party assists the ledger by processing macropayments faster.

[4] One deposit may back multiple payments; in particular, an honest customer may use a single deposit to back all of his payments.

can be detected; and (iii) if a merchant detects a double spend then he reports it, and doing so results in the revocation of the cheating customer's deposit. (Our cryptographic constructions will provide suitable mechanisms for these tasks.) We then ask: *how large must the deposit be in order to deter double spending?*

We provide a simple yet powerful analysis that answers this question under reasonable network behavior. Namely, let T denote the time it takes to catch a macropayment double spend (e.g., in Bitcoin one could take T to be the network's broadcast time). Within any period of time T, let A denote the maximum cumulative value of probabilistic payments and W the maximum cumulative value of macropayments; our analysis will show that imposing bounds on these quantities is *necessary*. To simplify discussions, we make the assumption that *only* macropayment (and not nullpayment) double spends are detectable; our analysis extends to the case where nullpayment double spends may also be detected eventually (see Remark 3.5).

Below we informally state our theorem, for simplicity in the special case where the macropayment value V and the payment probability p are fixed across all probabilistic payments, and all merchants share the same detection time T. The formal statement that we prove is in fact more general, because it applies even when these quantities are chosen dynamically and arbitrarily across different payments.

Theorem 1.1 (informal statement of Thm. 3.4)**.**
(a) If the deposit is at least W, then there is no worst-case utility gain in double spending.
(b) If the deposit is at least $(1-p)V + A$, then there is no average-case utility gain in double spending.
(c) Both bounds above are tight.

Our theorem has a simple interpretation: the required deposit amount equals the maximum financial activity that can happen within any time period of T. Namely, if macropayments have maximum total worth W within time T, the deposit must be at least W (w.r.t. worst-case utility); and if probabilistic payments have maximum total worth A within time T, the deposit must be at least $\approx A$ (w.r.t. average-case utility). Note that it is unsurprising that the two statements in the theorem depend on the two different quantities W and A, because they target different notions of utility; also note that, while one can take $pW \leq A$ without loss of generality, a bound on W does *not* always imply a bound on A (there could still be arbitrarily many probabilistic payments, though with extremely small probability).

But which of the two bounds should one use in practice? Naturally, the worst-case bound is safer than the average-case bound; however, an appropriate setting of W will be $\Omega(1/p)$ larger than A, which implies a substantial increase in the required deposit. The choice between the two depends on whether one cares about malicious customers that are lucky with even very small probability (as opposed to focusing on their average gains possibly across many deposits).

As already mentioned, bounding the value of probabilistic payments (via A) or macropayments (via W) within time T is *necessary* because our bounds are tight (i.e., there exist double-spending strategies that achieve them). In the "real world" these bounds may be imposed by the environment (e.g., limited network throughput), or the merchants (e.g., they accept up to a given number of payments within time T).

In terms of analysis, our proof shows that any additional utility gained via double spending must come from *macropayment* double spending. This may be surprising because, superficially, one may think that *nullpayment* double spending also contributes to additional utility; e.g., one may think that a malicious customer gains pV for every nullpayment double spend. This proposition is alarming: in the worst case there could be infinitely-many nullpayment double spends (which imply infinite additional utility); and in the average case there could be clever strategies that leverage double spends across multiple merchants to lower the probability of detection. We prove that this is not the case: we use a simulation argument to show that *the naive strategy of double spending as much as possible is the best strategy* (i.e., maximizes additional utility), both in the worst case and in the average case. In particular, we learn that the best strategy always leads to detection (after a time period of T) and that additional utility *is finite even in the worst case* (if W is finite). Details of our analysis are in Section 3.

We believe our theorem to be of independent interest because it applies to virtually any (centralized or decentralized) setting that enforces a deposit mechanism for offline payments. One such setting could be probabilistic *smart contracts* (an application suggested by [PS15, PS16]). A thorough understanding of the economic benefits of double spending is necessary to ensure that such smart contracts, as well as other applications, function as intended.

Example. As a demonstration, we invoke our theorem on parameters that could fit the application of *advertisement-free content delivery*, to see what conclusions our economic analysis gives us. Suppose that we consider a Bitcoin-like setting, where (i) transaction fees are typically a few cents; and (ii) we could take the detection time T to be, e.g., 20

minutes, which is typically two blocks (ideal block generation follows an exponential distribution with a mean of 10 minutes). Suppose further that we fix the deposit to be $D := \$200$ and the expected value of the probabilistic payment to be $\$0.1$ (similar size as a transaction fee); concentration bounds then suggest that, subject to the condition $pV = \$0.1$, good choices are $V := \$10$ and $p := 1\%$. Note that these settings imply that we can take W up to $D = \$200$ and A up to $D - (1-p)V = \$190.1$. Then our theorem implies that: (1) Even the *luckiest* double spending user has no extra utility gain if the cumulative value of macropayments every 20 minutes is less than $\$200$ (that is, the number of macropayments every 20 minutes is less than 20), regardless of how much nullpayment double spending occurred. (2) A double spending user has no extra utility gain on average if the cumulative value of probabilistic payments every 20 minutes is less than $\$190.1$ (that is, the number of probabilistic payments every 20 minutes is less than 1901).

1.1.2 Decentralized anonymous micropayments

We formulate the notion of a *decentralized anonymous micropayment* (DAM) scheme. This notion formalizes the functionality and security properties of an offline probabilistic payment scheme that enables parties with access to a ledger to conduct transactions with one another, directly and privately. To realize the requirements of our economic analysis, a DAM scheme enables parties to set up deposits, which are revoked when macropayments reveal that double spending has occurred. Crucially, the security guarantees of a DAM scheme guarantee anonymity not only across macropayments but also across nullpayments, so that even the "offline stream of payments" remains unlinkable.

We construct a DAM scheme and prove its security under specific cryptographic assumptions. Our two main building blocks are decentralized anonymous payment (DAP) schemes [BCG+14] and fractional message transfer schemes (see below).

Theorem 1.2 (informal statement of Thm. 7.1)**.** *Given a decentralized anonymous payment scheme and a fractional message transfer scheme (and other standard cryptographic primitives) there exists a DAM scheme.*

Formally capturing the notion of a DAM scheme and proving security of our construction was quite challenging due to the combination of rich functionality and strong anonymity guarantees. Parties can mint standard coins, deposits, or lottery tickets; they can withdraw deposits; they can pay each other with deterministic payments, switch coin types; they can also pay each other with probabilistic payments; they can revoke deposits of cheating parties — all of this while essentially revealing no information about origins, destinations, and amounts of money transfers. In particular, two features of our construction required particular attention: (1) revocation of an unknown cheating party's deposit when two macropayments with the same ticket are detected; and (2) monitoring of payment value rates (as required by our economic analysis) despite deposits being anonymous. Deterministic payments in our construction are non-interactive, while probabilistic payments consist of a 3-message protocol between a sender and a receiver; it is an interesting open question whether these can be made non-interactive as well.

We express the security of a DAM scheme via the ideal-world/real-world paradigm, specifying a suitable ideal functionality, and we prove our construction's security via a simulator against non-adaptive corruptions of parties. We consider security in the standalone setting, and leave security under composition to future work (that perhaps can build upon the work of [KMS+16]).

1.1.3 Fractional message transfer

A key ingredient in our construction of DAM schemes is *fractional message transfer* (FMT): a primitive that enables probabilistic message transmission between two parties, called the 'sender' and the 'receiver'. Informally, FMT works as follows: (i) the receiver samples a one-time key pair based on a transfer probability p; (ii) the sender uses the receiver's public key to encrypt a message m into a ciphertext c; (iii) the receiver uses the secret key to decrypt c, thereby learning m, but only with the pre-defined probability p (and otherwise learns no information about m).

We thus (1) formulate the notion of an *FMT scheme*, which formally captures the functionality and security of probabilistic message transmission, and (2) present an efficient construction that works for probabilities that are inverses of positive integers.

Theorem 1.3 (informal statement of Thm. A.3)**.** *In the random oracle model and assuming the hardness of* DDH *in prime-order groups, there exists an FMT scheme that works for transfer probabilities $p = 1/n$ with $n \in \mathbb{N}$. Moreover, the number of group elements and scalars in the public key and ciphertext is constant (independent of n); see Table 1.*

Our definition of FMT is closely related to *non-interactive fractional oblivious transfer* (NFOT), which was studied in the context of 'translucent cryptography' as an alternative to key escrow [BM89, BR99]. Namely, prior definitions target *one-way security*, which protects *random* messages. While one-way security suffices to encapsulate random secret keys (the setting of translucent cryptography), it does not suffice for probabilistically transmitting non-random messages (as needed in our construction). Therefore, our definition of an FMT scheme targets a fractional variant of *semantic security*, which we express via two properties: *fractional hiding* and *fractional binding*. Furthermore, since in our system any party can act as both sender and receiver, we require the FMT scheme to be *composable*. Our construction achieves this via simulation-extractability.

Our construction of FMT is loosely related to the constructions in [BM89, BR99], which (like our construction) build on the Elgamal encryption scheme [Elg85]. In fact, such constructions, if analyzed under the hardness of DDH rather than CDH, are likely to yield FMT according to our stronger definition. We did not carry out such an analysis, but instead chose to construct a scheme that is more efficient than prior work for the case of $p = 1/n$ (these probabilities suffice for our application); we assume hardness of DDH and work in the random oracle model in order to take advantage of certain Σ-protocols. See Table 1 for a comparison of our construction with prior work.

See Section 4 and Appendix A for more details.

scheme	**security**	**assumption**	**transfer probability**	**size of public key**		**size of ciphertext**		**# exponentiations**	
				group elts.	scalars	group elts.	scalars	to encrypt	to decrypt
[BM89]	one-way	CDH	$1/2$	2	—	2	—	2	1
[BR99, § 5.1]	one-way	CDH	$1/n$	n	—	2	—	2	1
[BR99, § 5.1]	one-way	CDH	$(n-1)/n$	n	—	2	—	2	1
[BR99, § 5.2]	one-way	CDH	a/n *	$2\log_2 n$	—	$2\log_2 n$	—	$4\log_2 n$	$2\log_2 n$
[BR99, § 5.3]	one-way	CDH	a/n	$a+n$	—	2	—	2	1
our FMT	semantic	DDH + RO	$1/n$	2	3	2	2	4	4

* n is restricted to be a power of 2.

Table 1: Comparison of prior NFOT schemes vs. our FMT scheme. All constructions assume a common random string.

1.2 Prior work on micropayment channels

Micropayment channels were introduced by Hearn and Spilman [HS12, Bit13], and further studied by Poon and Dryja [PD16] and Decker and Wattenhofer [DW15]. Roughly, a micropayment channel enables a sender and a receiver to set up a contract by way of an online (slow) transaction that escrows funds, after which the sender and receiver can update the contract, and thus the relative split of the escrowed funds, without recording the new contract on the blockchain. Thus payments can be made instantaneously. These can be dynamically combined to obtain multi-hop "payment channel networks" that go through several intermediaries, by using hashed timelock contracts; this technique amortizes the cost of setting up a new channel for new receivers. From the perspective of our work, micropayment channels have several limitations in terms of economics, functionality, and privacy.

Economic limitations of payment channels. First, payment channels in general require a channel to be established in advance with a party: payments are only instantaneous with advanced preparation. To alleviate this constraint, payment channel networks allow transactions with arbitrary new parties provided there exists a path of existing channels between the payer and payee.

Such networks have limitations. First, considerable capital is escrowed in the many pairwise channels forming the network. The capital requirements may exceed those required for deposits in probabilistic micropayments. Both settings require escrowed funds proportional to a user's economic activity (either for the double spend deposit or the "last mile" channel between the user and the payment network), but payment channel networks escrow similar amounts in each edge of the network. Second, a variety of pressures, including minimizing the capital escrowed, may centralize such networks into a hub-and-spoke model.

Privacy limitations of payment channels. Payment channels reveal to the world that a given pair of parties have a channel between them, the opening value of that channel, and the final closing value. More importantly, especially for

applications like advertisement-free content delivery, payment channels provide no privacy between the parties on the channel: if Alice pays say Wikipedia every time she views a page, then each of those views is linked to the channel she established just as effectively as if she had a tracking cookie in her browser.

Attempts to add privacy, either from intermediate nodes in the network [HAB+16] or from recipients and intermediaries [GM16], to payment channels hit some seemingly fundamental limitations of the payment channel setting. First, the anonymity set when paying a given receiver is composed only of those users who have opened channels with the receiver. This is likely far smaller than the global anonymity set provided by probabilistic payments. Moreover, the receiving party can arbitrarily reduce the anonymity set further by closing channels. This leaves open a range of attacks that are not present in a system with a global anonymity set.

Finally it is unclear if non-hub-and-spoke private payment networks are scalable or can provide privacy for payment values from intermediary nodes in the network. When a payment is made via two intermediaries (i.e. $A \to I_1 \to I_2 \to B$), some combination of I_1 and I_2 must know the balance of their pairwise channel at any given time or they could not close the channel. Thus the value of any payment relayed through multiple parties cannot be completely private. Moreover, discovering a multi-hop route between two parties in a diverse and large network without leaking any identifying information seems costly at scale. While [GM16] extend their point-to-point channel protocol to a hub-and-spoke model that alleviates both these concerns, such a network is inherently centralized.

1.3 Roadmap

The remainder of the paper is organized as follows. In Section 2 we describe the intuition and techniques behind our results. In Section 3 we present our economic analysis of double spending. In Section 4 we present our result on fractional message transfer. In Section 5 we recall the notion of a DAP scheme from [BCG+14]. In Section 6 we present our definition of a DAM scheme. In Section 7 we present our construction of a DAM scheme.

2 Techniques

We discuss the intuition and techniques behind our results, first for our cryptographic construction (Section 2.1) and then for our economic analysis of double spending (Section 2.2).

2.1 Constructing decentralized anonymous payments

We discuss our design of a decentralized anonymous micropayment (DAM) scheme via a sequence of candidate constructions, each fixing problems of the previous one; the last one is a sketch of our construction.

2.1.1 Attempt 1: non-anonymous probabilistic payments + DAP

We begin with a natural candidate construction for a DAM scheme. The idea is to combine two primitives, one providing probabilistic payments and the other anonymity. For example, consider: (1) the scheme `MICROPAY1` of [PS15], which provides probabilistic payments for Bitcoin;[5] and (2) a *decentralized anonymous payment* (DAP) scheme [BCG+14], which provides privacy-preserving payments for Bitcoin-like currencies.

To make `MICROPAY1` privacy-preserving, we could try to replace its Bitcoin payments with DAP payments, which hide the payment's origin, destination, and amount. Thus, when a probabilistic payment goes through, and the corresponding DAP (macro-)payment is broadcast, others cannot learn this information about the payment. However, this idea does *not* provide the strong anonymity guarantees that we seek, as we now explain.

Problem: not fully anonymous. Despite the anonymity guarantees provided by the DAP scheme, merchants still learn information about their customers' spending habits. Each DAP payment includes a unique serial number corresponding to the underlying "coin" that was spent by that payment; this is used to prevent double spending of DAP coins. In the above proposal, the customer sends the merchant this serial number regardless of whether the payment becomes a nullpayment or a macropayment. Since the same underlying DAP payment and serial number are used across multiple probabilistic payments (to amortize fees), this compromises customer anonymity because a merchant learns (1) which (macro or null) payments to him were made with the same escrow; and (2) which macropayments to other merchants were made with an escrow used for payments to him. This compromise in anonymity gets even worse if merchants share such information with one another.

Moreover, recall (from Section 1.1) that it is not possible to prevent double spending in the setting of offline probabilistic payments. Pass and Shelat note this in the full version of their paper [PS16], and propose adding a 'penalty escrow' to the scheme `MICROPAY1`; the escrow is burned upon evidence of double spending. But observe that anonymity for penalty escrows poses a similar challenge: to prove that a penalty escrow is unspent, a merchant reveals its serial number, once again enabling merchants to link probabilistic payments by learning about their escrows.

Overall, while the above ideas do achieve unlinkability of macropayments, customers have little meaningful privacy until nullpayments and escrows are also unlinkable.

2.1.2 Attempt 2: commit to DAP payment + probabilistic opening + private deposit coins

One way to address the anonymity problems of the previous attempt is to ensure that the merchant learns the serial number only when the payment turns into a macropayment (and, conversely, learns nothing otherwise). Then, to enable the aforementioned penalty escrow mechanism, a customer creates a special 'deposit' coin.

Then, the modified protocol works as follows: (1) the customer sends to the merchant a commitment to a DAP payment and to a 2-out-of-n share of the deposit serial number; (2) the customer and merchant engage in a protocol that opens the commitment with probability p (opening thus corresponds to a macropayment, and not opening corresponds to a nullpayment); (3) when publishing a macropayment to the ledger, the merchant also publishes the secret share.

The probabilistic opening hides the serial number of the coin in the DAP payment until a macropayment occurs, and the secret share hides the deposit serial number until a macropayment double spend occurs. To punish a double spending customer, the merchant obtains (from the network or from the ledger) two secret shares of the deposit serial

[5] The other two schemes of [PS15] rely on a trusted third party for assisting the ledger in processing payments or resolving disputes, which makes it harder to achieve strong anonymity guarantees.

number from two macropayments and reconstructs the serial number. He then publishes this to the ledger, thereby blacklisting the deposit.

One issue that must be addressed is ensuring that the secret shared deposit serial number corresponds to a valid deposit. To do this, first notice that there are two kinds of blacklisted deposits: those whose serial number appears on the ledger (in previous 'punish' transactions), and those that have been revoked in the current epoch. The serial numbers of the latter kind are broadcast across the network, but have not yet appeared on the ledger.

To prevent users from using blacklisted deposits of the first kind, a customer must prove to the merchant that his deposit's serial number does not appear on the ledger (this can be done efficiently [MRK03]). To prevent use of deposits of the second kind, customers must also send to the merchant a tag derived from the deposit's serial number. Since anyone with access to this serial number can compute this tag, merchants can deduce if a deposit has been revoked by checking if this tag has been computed with a blacklisted deposit's serial number. The customer accompanies the tag with a zero-knowledge proof that the deposit used for this tag is consistent with the share inside the commitment.

The aforementioned proposal, however, is still vulnerable to attacks.

Problem: front-running deposit revocation. While deposits are intended to deter double spending, customers may try to withdraw a deposit before it is blacklisted, thereby rendering punishment ineffective.

Problem: merchant aborts. At the end of the commitment opening protocol, the merchant can refuse to inform the customer of whether or not the commitment was opened. This poses a problem for the customer because if the commitment was in fact opened, the merchant has learned the serial number and a share of the deposit, enabling him to: (i) track the customer and learn when they spend the coin with another merchant, and (ii) revoke the customer's deposit after the (honest) customer next spends the coin, with another merchant or the same one.

2.1.3 Outline of our construction

The deposit mechanism described so far is insufficient to deter double spending. The problem is that there is no restriction on how and when coins used for probabilistic payments and for deposits can be transferred; in particular, a cheating customer can double spend these back to himself while at the same time engaging in a probabilistic payment with a merchant. We address this problem by (i) partitioning coins into different types depending on their different uses, and (ii) restricting transfers between coins depending on their types. We now outline how we carry out this plan.

First, we extend the notion of a DAP scheme to allow users to associate public and private information strings when minting a coin. Users can now store a coin's type in its public information string, and we allow three types of coins: in addition to the "standard" coin type, we introduce deposits and tickets. A ticket is bound to a deposit by storing the deposit inside the ticket's private information string. We thus have the following semantics:

- *Coins* are used for deterministic DAP payments (whose processing fees are not amortized).
- *Deposits* are used to back tickets and are revoked when two macropayments using the same ticket are detected.
- *Tickets* are used for probabilistic payments; every ticket is bound to a single deposit at minting time, and can be spent provided that the associated deposit is valid (i.e., has not be transferred to a coin, or revoked).

We also restrict the set of possible transactions depending on the types of coins involved, as follows.

- *Transactions with coins:* Coins can be used to create other coins, deposits, or tickets. In particular, coin-to-coin transactions preserve the deterministic payment functionality of the underlying DAP scheme.
- *Transactions with deposits:* Deposit-to-coin transactions let customers withdraw deposits, though not immediately, since these transactions become active only after an *activation delay* Δ_{w} that is a parameter of the system.
- *Transactions with tickets:* Ticket-to-coin transactions enable probabilistic payments; they are associated with a secret share of the ticket's deposit and with a deposit-derived tag that allows merchants to detect the validity of the ticket's deposit. Ticket-to-ticket transactions omit the secret share and tag and (like deposit-to-coin transactions) become active only after an *activation delay* Δ_{r} that is a parameter of the system.

Restrictions on inter-type transactions are achieved via a *pour predicate* that checks that input and output coin types satisfy the above restrictions. Having made these modifications, we can now resolve the issues of the previous proposal.

Preventing deposit theft. Deposit-to-coin transactions now have a delayed activation, so customers can no longer withdraw deposits before they are blacklisted, as merchants have enough time to post deposit revocations to the ledger.

Recovering from merchant aborts. Since we cannot know what is the utility gain of a merchant for learning about the spending patterns of a customer, we cannot effectively deter merchant aborts by economic means. Instead, at the end

of our commitment opening protocol, we require the merchant to prove to the customer whether or not he could open the commitment. If the merchant fails to do so, we allow customers to "refresh" their tickets by creating a ticket-to-ticket payment to themselves. Since the new ticket has a different serial number that merchants have not yet seen, they cannot track the new ticket's transaction history. Finally, since ticket-to-ticket transactions become active only after a delay, the new tickets cannot be spent immediately, thus allowing merchants to post macropayments over the old ticket.

The above sketch omits many technical details, including how a DAM scheme interacts with the economic analysis. See Section 6 for the definition of a DAM scheme, Section 7 for our construction of a DAM scheme, and Appendix C for its security proof.

2.2 Intuition for our economic analysis of double spending

Our economic analysis characterizes the additional utility that customers can gain by double spending in offline probabilistic payments. We discuss the intuition for the analysis via an example; details of the analysis are in Section 3 (the formal statement is Theorem 3.4). Recall that we assume that: (i) every probabilistic payment is backed by an advance deposit, and (ii) macropayment double spends are detected within time T, and result in deposit revocation.

At a high level, the deposit must be at least as large as the additional utility that a malicious customer gains by double spending until that deposit is revoked; additional utility occurs when the customer double spends, and accumulates until cheating is detected and every merchant has blacklisted the customer. If we can bound the value of payments in this period of time, then we can derive a corresponding bound on the additional utility gained, and thus bound the deposit.

A naive analysis, however, yields an impractically large bound, because the natural definition of "additional utility" is too coarse. We illustrate this issue via an example: a malicious customer $\tilde{\mathsf{C}}$ selects two merchants $\mathsf{M}_1, \mathsf{M}_2$, and uses the same "lottery ticket" to conduct parallel probabilistic payments $\widetilde{\mathsf{pay}}_1, \widetilde{\mathsf{pay}}_2$ to $\mathsf{M}_1, \mathsf{M}_2$ respectively. The merchants cannot immediately detect that $\tilde{\mathsf{C}}$ is cheating because $\tilde{\mathsf{C}}$ is indistinguishable from an honest user so far. If both $\widetilde{\mathsf{pay}}_1$ and $\widetilde{\mathsf{pay}}_2$ become macropayments, which happens with probability p^2, then the merchants (eventually) catch $\tilde{\mathsf{C}}$ cheating, and revoke $\tilde{\mathsf{C}}$'s deposit of value D. Consider the following two analyses.

(i) A naive analysis. The malicious customer $\tilde{\mathsf{C}}$ earns an additional utility of pV compared to an honest customer, and is caught and punished by D with probability p^2. Hence, to deter $\tilde{\mathsf{C}}$ from cheating, the deposit amount should be such that $p^2 D > pV$, which is equivalent to $D > V/p$.

(ii) A better analysis. The average-case utility $\mathbb{E}[\mathcal{U}(\mathsf{C})]$ of an honest customer C for any probabilistic payment is zero: C gains pV with probability $1-p$, and $pV - V$ with probability p. Instead, the utility $\mathcal{U}(\tilde{\mathsf{C}})$ of the malicious customer $\tilde{\mathsf{C}}$ has four cases, as given in Table 2; also, $\tilde{\mathsf{C}}$ is caught and punished by D with probability p^2. Thus, the deposit amount should be such that $p^2 D > \mathbb{E}[\mathcal{U}(\tilde{\mathsf{C}})] = 2pV - (1-(1-p)^2)V$, which is equivalent to $D > V$.

$\widetilde{\mathsf{pay}}_1$ \ $\widetilde{\mathsf{pay}}_2$	null	macro
null	$2pV$	$2pV - V$
macro	$2pV - V$	$2pV - V$

Table 2: Utility $\mathcal{U}(\tilde{\mathsf{C}})$ of the malicious customer $\tilde{\mathsf{C}}$.

pay_1 \ pay_2	null	macro
null	$2pV$	$2pV - V$
macro	$2pV - V$	$2pV - 2V$

Table 3: Utility $\mathcal{U}(\mathsf{C})$ of the honest customer C.

How do the two analyses differ? The first analysis states that the deposit amount D must be greater than V/p while the second states that it must be greater than V, which is a much smaller lower bound. This is because the first analysis adopted an intuitive, but coarse, definition of additional utility, which did not consider the fact that a malicious customer does not gain additional utility unless two macropayments with the same ticket occur. Indeed, the utility $\mathcal{U}(\mathsf{C})$ of an honest user C that uses two *different* tickets to make two parallel probabilistic payments $\mathsf{pay}_1, \mathsf{pay}_2$ is in Table 3. By comparing $\mathcal{U}(\tilde{\mathsf{C}})$ and $\mathcal{U}(\mathsf{C})$, one can see that the utility function differs *only when two macropayments occur*, where, if there is no deposit/punishment, $\tilde{\mathsf{C}}$ gains extra utility of V by paying only one macropayment instead of paying two as C does. In sum, any additional utility gained via double spending *must come from macropayment double spends*.

Towards a general analysis. The above discussion suggests that the additional utility of $\tilde{\mathsf{C}}$, which we denote by

$\mathcal{U}'(\tilde{\mathsf{C}})$, should be defined as follows:

$$\mathcal{U}'(\tilde{\mathsf{C}}) := \begin{cases} V & \text{if } \widetilde{\mathsf{pay}}_1, \widetilde{\mathsf{pay}}_2 \text{ are macropayments} \\ 0 & \text{otherwise} \end{cases} .$$

More generally, the additional utility of any malicious customer $\tilde{\mathsf{C}}$ is the extra gain compared to an honest customer achieving the same outcome. This can be computed by considering an honest customer C that simulates the behavior of $\tilde{\mathsf{C}}$ while only using unspent tickets; the extra gain arises from the fact that C has "paid" for these other unspent tickets while $\tilde{\mathsf{C}}$ has not. By understanding the maximum of this refined notion of additional utility we can derive the minimum amount of deposit needed such that, for any double spending attack, there is a non-double-spending strategy that achieves better utility, in the worst-case and in the average-case respectively. See Section 3 for a formal argument of this intuition, as well as a discussion of the implications of our economic analysis.

3 Economic analysis of double spending for offline probabilistic payments

We provide the economic analysis that characterizes the additional utility that can be gained by double spending via offline probabilistic payments. This section is organized as follows. First, we informally describe dynamics that model offline probabilistic payments (Section 3.1). Then, we define a formal game that captures these dynamics and analyze this game (Section 3.2). Finally, we discuss the interpretation and consequences of our economic analysis (Section 3.3).

3.1 Informal description of payment dynamics

We informally describe the dynamics of arbitrary probabilistic payments from customers to merchants. A concrete example is the setting of *advertisement-free Internet*: a customer is a user surfing the Internet; a merchant is a web server; every HTTP request by a user to a web server is accompanied by a probabilistic payment from that user to the web server (to buy an ad-free HTTP response).

Abstraction of probabilistic payments. A probabilistic payment is an interactive protocol between a customer and a merchant. The customer's input is a *ticket* $\mathbf{t} = (\mathsf{t}, p, V, \mathbf{d})$ where $\mathsf{t} \in \{0,1\}^*$ is the unique *ticket identifier*, $p \in [0,1]$ is the *payment probability*, $V \in \mathbb{R}_{\geq 0}$ is the *macropayment value*, and $\mathbf{d} = (\mathsf{d}, D)$ is the *deposit*, which consists of a unique *deposit identifier* $\mathsf{d} \in \{0,1\}^*$ and a *deposit value* $D \in \mathbb{R}_{\geq 0}$. Informally, the customer first convinces the merchant that the deposit is not "invalid", and then the customer pays V to the merchant with probability p. The two outcomes are called a *nullpayment* and a *macropayment*, and involve different protocol outputs.

Detectable double spends. At any moment in time, a deposit is in one of two states: *valid* or *invalid*. Each deposit is initially valid. When two macropayments occur on the same ticket $\mathbf{t}$, the associated deposit $\mathbf{d}$ becomes invalid, once and for all. We call this event a *macropayment double spend*, and we assume that, in this case, the underlying probabilistic payment protocol enables merchants to eventually learn that $\mathbf{d}$ (more precisely, its identifier) has become invalid;[6] we denote by T_{M} the time for merchant M to learn this from the moment the macropayment double spend occurred. The fact that $\max_{\mathsf{M}} T_{\mathsf{M}} > 0$ is the fundamental reason that allows a malicious customer to gain any additional utility.

Finally, we make the simplifying assumption that, while macropayment double spends are detectable, nullpayment double spends are undetectable. Our analysis does extend to the case where (not necessarily all) nullpayment double spends are also detectable; see Remark 3.5.

Honesty of merchants. We assume that *merchants behave honestly*. Thus, every merchant (a) rejects aborted payments (e.g., due to invalid deposits); (b) honors successful payments (e.g., replies with an ad-free HTTP response) regardless of whether the payment resulted in a nullpayment or macropayment; (c) reports detected double spends; more generally, (d) follows the probabilistic payment protocol (e.g., uses fresh randomness in each instance of the protocol, broadcasts any messages to all other merchants as instructed, and so on).

In principle, merchants may deviate from the aforementioned honest behavior in a variety of ways. For instance, a merchant may "honor" an aborted payment (e.g., regardless of the validity of the customer's deposit); or the merchant may not honor a successful payment (e.g., does not reply to the HTTP request); or the merchant may abort and prevent the customer from learning the payment's outcome; or the merchant may not report a detected double spend.

However, we assume that all merchants behave honestly because the only incentive for a merchant to deviate comes from colluding with malicious customers, and we cannot prevent such collusions. Indeed, if a merchant does not collude with any malicious customer, then for the merchant it is individually rational to behave honestly, because: (i) some malicious merchant behavior (e.g., "honoring" an aborted payment, or using correlated randomness across payments) does not increase the merchant's utility; (ii) other malicious merchant behavior (e.g., not honoring a successful payment) decreases the customer's utility, but taking into account this possibility does not affect a customer's maximum additional utility (the quantity we study) and ruling it out significantly simplifies the analysis. However, a malicious customer could convince a merchant to *not* report a double spend by offering side payments as compensation; if the merchant has already replied to the customer's payment then this collusion may indeed be economically attractive, but we cannot

[6] Exactly *how* merchants learn $\mathbf{d}$'s identifier depends on the details of a construction, and is orthogonal to our economic analysis; ditto for exactly how the monetary funds escrowed in $\mathbf{d}$ are revoked after $\mathbf{d}$ becomes invalid. (See Section 7 for how we do so in the particular case of our construction; informally, we rely on the fact that a macropayment to a merchant reveals a ticket's identifier, and two macropayments on the same ticket allow a merchant to deduce the identifier of the ticket's deposit.)

systematically prevent such side payments in all applications. (In the setting of micropayments, V is small so a merchant may prefer to see the malicious customer punished, after losing V, rather than receiving compensation.)

Honest vs. malicious customers. Our goal is to characterize the additional utility obtained by any malicious customer, when compared to what is possible by honest customers. We now discuss both kinds of customers.

Honest customers. For an honest customer, a ticket $\mathtt{t}$ is in one of three states: it is *spent* if a probabilistic payment on it has resulted in a macropayment; otherwise, it is *occupied* if it is being used in a probabilistic payment; otherwise, it is *unspent* (i.e., it never resulted in a macropayment, nor is it being used in a probabilistic payment).

At any moment in time, an honest customer may select any number of merchants, and initiate any number of probabilistic payments in parallel to every one of them. Each probabilistic payment uses a distinct unspent ticket, which immediately becomes occupied, and at the end of the payment protocol becomes either unspent or spent. The selected tickets may or may not have different deposits that back them; deposits are never invalidated for honest customers. In sum, an honest customer maintains the invariant that an occupied ticket does not participate in more than one payment at a time, and a spent ticket does not participate in future payments.

Malicious customers. A malicious customer may deviate from the aforementioned honest behavior in a variety of ways, as we now describe. Like an honest customer, a malicious customer owns an arbitrary number of tickets and deposits; unlike an honest customer, a malicious customer may use an occupied ticket in multiple payments, or may use a spent ticket in future payments (hence, a ticket of a malicious customer could be in *both spent and occupied states at the same time*). We give some examples of malicious behavior.

- **One-ticket-one-merchant attack.** A malicious customer $\tilde{\mathsf{C}}$ has a ticket $\mathtt{t}$ and selects a merchant M; then $\tilde{\mathsf{C}}$ initiates multiple probabilistic payments to M in parallel, and continues using the same ticket $\mathtt{t}$ even after it is spent. The merchant M cannot detect that $\tilde{\mathsf{C}}$ is cheating until M receives two macropayments relative to the same ticket $\mathtt{t}$.
- **One-ticket-multiple-merchant attack.** A malicious customer $\tilde{\mathsf{C}}$ has a ticket $\mathtt{t}$ and selects two merchants $\mathsf{M}_1, \mathsf{M}_2$; then $\tilde{\mathsf{C}}$ conducts a sequence of probabilistic payments to M_1, using $\mathtt{t}$ until it is spent to M_1. In parallel, $\tilde{\mathsf{C}}$ adopts the same strategy with M_2, until $\mathtt{t}$ is spent to M_2. Observe that $\tilde{\mathsf{C}}$ acts like an honest customer to M_1 and M_2 individually; hence, the two merchants cannot detect that $\tilde{\mathsf{C}}$ is cheating until they communicate.
- **Multiple-ticket-multiple-merchant attack.** More generally, a malicious customer $\tilde{\mathsf{C}}$ has multiple tickets $\mathtt{t}_1, \mathtt{t}_2, \ldots$ and selects multiple merchants $\mathsf{M}_1, \mathsf{M}_2, \ldots$; then $\tilde{\mathsf{C}}$ conducts a sequence of probabilistic payments to M_1, using $\mathtt{t}_1$ until it is spent to M_1. Then $\tilde{\mathsf{C}}$ switches to $\mathtt{t}_2$ and continues making probabilistic payments to M_1 until $\mathtt{t}_2$ is spent. The customer $\tilde{\mathsf{C}}$ continues in this way until all the tickets are spent to M_1. In parallel, $\tilde{\mathsf{C}}$ adopts the same strategy with every other merchant. Observe again that $\tilde{\mathsf{C}}$ acts like an honest customer to each merchant individually; hence, the merchants cannot detect that $\tilde{\mathsf{C}}$ is cheating until they communicate.

Recall that, no matter what a malicious customer does, whenever two macropayments relative to the same ticket $\mathtt{t}$ occur, the deposit of $\mathtt{t}$ becomes invalid, and eventually (after at most time $\max_{\mathsf{M}} T_{\mathsf{M}}$) all merchants learn about this.

Towards a formal game. The above discussion leads us to the following informal description of arbitrary dynamics of probabilistic payments from a potentially-malicious customer to honest merchants; this description is only an intermediate step that we provide for intuition, because we formally define an abstract game in Section 3.2 below.

For each time t, let $\mathcal{I}(t)$ denote the set of deposit identifiers of invalid deposits at time t. This set is not maintained by anyone: by definition it contains the correct identifiers at any time. It is public and, hence, known to the customer.

Suppose that a customer initiates a probabilistic payment with merchant M at time t, using a ticket $\mathtt{t} = (\mathtt{t}, V, p, (\mathsf{d}, D))$. If $\mathsf{d} \in \mathcal{I}(t - T_{\mathsf{M}})$ (the deposit identifier belongs to an invalid deposit) then the payment aborts. Otherwise, (i) with probability $1 - p$, both parties receive the output `null`; (ii) with probability p, both parties receive the output `macro`.

Crucially, the decision of whether a payment aborts depends only on the global information from T_{M} units of time "into the past", because, in the worst case, there is a delay of T_{M} for merchant M to learn that a deposit has been invalidated. Of course, the merchant M may happen to learn this information faster than that; though modeling this fact does not ultimately change the maximum additional utility, so we ignore this for simplicity. This means that all merchants "behave the same" and thus we replace them with a single abstract player, 'Nature', in the next section.

Note that a construction of a probabilistic payment should also involve a check of whether the deposit value D is "large enough" to back the payment (as informed by our economic analysis). We ignore this check (and how it can be performed) because it is irrelevant to the economic analysis. (But see Section 7 for how our construction does it.)

3.2 The game and its analysis

We define a single-player game against Nature that captures the dynamics described in Section 3.1, namely, the dynamics of a customer $\tilde{\mathsf{C}}$ conducting arbitrary probabilistic payments with all merchants. We prove tight bounds on $\tilde{\mathsf{C}}$'s additional utility, in the worst case and in the average case. Note that, due to the additive nature of utility, we only need to analyze $\tilde{\mathsf{C}}$'s additional utility *per deposit*; hence, we restrict $\tilde{\mathsf{C}}$ to backing all his probabilistic payments with a single deposit.

As mentioned in Section 1.1.1, our analysis involves two parameters A and W, which denote the (per-deposit) maximum value of probabilistic payments and of macropayments, within any "detection time period". More precisely, let T_{M} denote the time for a merchant M to detect a detectable double spend, and let $\mathfrak{a}_{\mathsf{M}}(t)$ be the (cumulative) value of probabilistic payments accepted by M within the time period $[t, t+T_{\mathsf{M}}]$; similarly, let $\mathfrak{w}_{\mathsf{M}}(t)$ be the (cumulative) value of macropayments accepted by M within the time period $[t, t+T_{\mathsf{M}}]$. The parameters A and W are defined as $\max_t \sum_{\mathsf{M}} \mathfrak{a}_{\mathsf{M}}(t)$ and $\max_t \sum_{\mathsf{M}} \mathfrak{w}_{\mathsf{M}}(t)$ respectively. We defer to Section 3.3 a discussion of the interpretation of these parameters, and for now we focus on analyzing the additional utility in terms of these.

We argue that it suffices to study $\tilde{\mathsf{C}}$'s additional utility across merchants within a certain time period, and to consider only probabilistic payments that use spent tickets.

- *Starting point.* It suffices to analyze $\tilde{\mathsf{C}}$'s additional utility from the first time when two macropayments occur relative to the same ticket; denote by $\widetilde{\mathsf{pay}}$ the payment among these that terminates later (if they terminate simultaneously then break ties arbitrarily). Indeed, recall that $\tilde{\mathsf{C}}$'s additional utility is the extra gain compared to any honest customer achieving the same outcome. So consider the honest customer C that uses unspent tickets for every probabilistic payment that terminates before $\widetilde{\mathsf{pay}}$ does: the utilities up to then for $\tilde{\mathsf{C}}$ and C are the same. Thus, we only need to consider $\tilde{\mathsf{C}}$'s additional utility from when $\widetilde{\mathsf{pay}}$ terminates.

- *Ending point.* It suffices to analyze $\tilde{\mathsf{C}}$'s additional utility from when $\widetilde{\mathsf{pay}}$ terminates until when every merchant M has detected $\tilde{\mathsf{C}}$'s cheating. Indeed, $\widetilde{\mathsf{pay}}$ is a detectable double spend, so within time T_{M} merchant M detects $\tilde{\mathsf{C}}$'s cheating (i.e., has learned that $\tilde{\mathsf{C}}$'s deposit is invalid) and will not accept $\tilde{\mathsf{C}}$'s probabilistic payment anymore. Moreover, $\tilde{\mathsf{C}}$'s deposit is eventually revoked.

- *Which payments.* It suffices to consider every probabilistic payment that terminates within the aforementioned time period and uses a ticket that is spent before the termination of that payment (if multiple payments terminate simultaneously then pick an arbitrary termination order for them). Throughout this section we say that these probabilistic payments *use spent tickets*, and say that the other probabilistic payments *use unspent tickets*. Indeed, consider again the honest customer C that uses unspent tickets for every probabilistic payment: the utilities for $\tilde{\mathsf{C}}$ and C are the same on probabilistic payments that use unspent tickets.

In conclusion, we only need to worry about $\tilde{\mathsf{C}}$'s additional utility from when $\widetilde{\mathsf{pay}}$ terminates until when every merchant has detected $\tilde{\mathsf{C}}$'s cheating, and it suffices to consider only probabilistic payments that use spent tickets.

Suppose that during this time period $\tilde{\mathsf{C}}$ has finished $C+1$ probabilistic payments, including $\widetilde{\mathsf{pay}}$, using spent tickets: $\widetilde{\mathsf{pay}}$ is fixed to be a macropayment, while the remaining C payments are probabilistic (i.e., turn into nullpayments or macropayments with the appropriate probability). Perhaps $\tilde{\mathsf{C}}$ only made $C+1$ payments, or perhaps the merchants accepted only the first $C+1$ and rejected the rest due to invalid or insufficient deposit. (We assume $C < \infty$ for ease of exposition, but we could replace C with ∞ and our analysis would still hold.) Either way, note that $\tilde{\mathsf{C}}$ may select the payment probability and macropayment value of a probabilistic payment based on the outcomes of prior probabilistic payments. Below we define a game that captures these payments.

Definition 3.1. *Consider the following single-player game against Nature.*

- *The set of randomness choices is $[0,1]^C$; Nature samples $\boldsymbol{\lambda}$ uniformly at random from $[0,1]^C$. We denote by $\boldsymbol{\lambda}_{<i}$ the first $(i-1)$ coordinates of $\boldsymbol{\lambda}$ (and define $\boldsymbol{\lambda}_{<0}$ and $\boldsymbol{\lambda}_{<1}$ to be the empty string).*

- *The player strategies Σ consist of tuples $\boldsymbol{\sigma} = (p_i, V_i)_{i=0}^{C}$ consisting of computable functions that, based on Nature's randomness choice, output parameters for all the probabilistic payments. More precisely, for each i, $p_i(\boldsymbol{\lambda}_{<i}) \in [0,1]$ is the payment probability of the i-th probabilistic payment, and $V_i(\boldsymbol{\lambda}_{<i}) \in \mathbb{R}_{\geq 0}$ is its macropayment value.*

The game proceeds as follows. The player selects a strategy $\boldsymbol{\sigma} \in \Sigma$; afterwards, Nature samples $\boldsymbol{\lambda}$, whose coordinates are revealed to the player round by round. More precisely, the game is played in rounds, as follows: in round i, the player learns $\boldsymbol{\lambda}_{<i}$, and conducts a probabilistic payment (using a spent ticket) with payment probability $p_i(\boldsymbol{\lambda}_{<i})$ and macropayment value $V_i(\boldsymbol{\lambda}_{<i})$. The outcome of the i-th round is given by the indicator $\mathbb{I}[\lambda_i \leq p_i(\boldsymbol{\lambda}_{<i})]$, stating whether the payment resulting in a macropayment (the indicator equals 1) or nullpayment (the indicator equals 0).

Observe that *all* strategies in the above game are double-spending strategies: as discussed, it suffices to consider only probabilistic payments that use spent tickets. We now turn to define additional utility. Comparing an honest customer with a malicious one, we observe that any additional utility comes only from macropayments that involve spent tickets. More precisely, the first such macropayment (which is $\widetilde{\mathsf{pay}}$) contributes additional utility V_0 and, after that, if the i-th probabilistic payment results in a macropayment then additional utility increases by $V_i(\boldsymbol{\lambda}_{<i})$. As for nullpayments, neither an honest nor a malicious customer loses tickets, hence additional utility does not increase. Therefore, we define additional utility as follows.

Definition 3.2. *The* **additional utility** *of a strategy $\boldsymbol{\sigma} \in \Sigma$ on randomness $\boldsymbol{\lambda} \in [0,1]^C$ is*

$$\mathcal{U}'_{\boldsymbol{\lambda}}(\boldsymbol{\sigma}) := V_0 + \sum_{i=1}^{C} \mathbb{I}[\lambda_i \leq p_i(\boldsymbol{\lambda}_{<i})] V_i(\boldsymbol{\lambda}_{<i}) \ .$$

(Additional utility is a random variable, as it depends on Nature's randomness $\boldsymbol{\lambda}$, which is a random variable.)

We analyze the *maximum* additional utility achievable by any strategy, in the worst case and in the average case, for the game from Definition 3.1; these maximum values bound from below the required deposit value D (for the goal of deterring double spending). Below we define two subsets of strategies in which the bounds A or W are respected. (Note that if $C < \infty$, then $(\min\{p_i\}_{i=0}^{C}) \cdot W \leq A$ so that if A is bounded then so is W.)

Definition 3.3. *We define the following two sets of strategies, which respectively capture the condition that the total worth of probabilistic payments is at most A and the total worth of macropayments is most W:*

$$\Sigma_A^{\mathsf{pp}} := \left\{ \boldsymbol{\sigma} \in \Sigma : \forall\, \boldsymbol{\lambda},\ p_0 V_0 + \sum_{i=1}^{C} p_i(\boldsymbol{\lambda}_{<i}) V_i(\boldsymbol{\lambda}_{<i}) \leq A \right\} \ ,$$

$$\Sigma_W^{\mathsf{mp}} := \left\{ \boldsymbol{\sigma} \in \Sigma : \forall\, \boldsymbol{\lambda},\ V_0 + \sum_{i=1}^{C} \mathbb{I}[\lambda_i \leq p_i(\boldsymbol{\lambda}_{<i})] V_i(\boldsymbol{\lambda}_{<i}) \leq W \right\} \ .$$

We now state and prove our worst-case and average-case bounds on additional utility. (Recall that, by Yao's minimax principle, it suffices to consider only deterministic strategies [Yao77], and thus we ignore randomized ones.)

Theorem 3.4 (formal statement of Thm. 1.1)**.** *For the game described above, the following holds.*

(a) WORST CASE: *for every randomness choice $\boldsymbol{\lambda} \in [0,1]^C$ and strategy $\boldsymbol{\sigma} \in \Sigma_W^{\mathsf{mp}}$, it holds that $\mathcal{U}'_{\boldsymbol{\lambda}}(\boldsymbol{\sigma}) \leq W$.*

(b) AVERAGE CASE: *for every strategy $\boldsymbol{\sigma} \in \Sigma_A^{\mathsf{pp}}$, it holds that $\mathbb{E}_{\boldsymbol{\lambda}}\left[\mathcal{U}'_{\boldsymbol{\lambda}}(\boldsymbol{\sigma})\right] \leq (1-p_0)V_0 + A$.*

(c) Both bounds are tight.

Proof. We prove the three statements in order.

Part (a). By definition of Σ_A^{pp} (see Definition 3.3), for every randomness choice $\boldsymbol{\lambda} \in [0,1]^C$ and strategy $\boldsymbol{\sigma} \in \Sigma_W^{\mathsf{mp}}$, it holds that $V_0 + \sum_{i=1}^{C} \mathbb{I}[\lambda_i \leq p_i(\boldsymbol{\lambda}_{<i})] V_i(\boldsymbol{\lambda}_{<i}) \leq W$; but the quantity on the left-hand side of the inequality is $\mathcal{U}'_{\boldsymbol{\lambda}}(\boldsymbol{\sigma})$ (see Definition 3.2), and the claimed statement follows.

Part (b). Recall that Nature samples $\boldsymbol{\lambda}$ uniformly at random from $[0,1]^C$, so the coordinates of $\boldsymbol{\lambda}$ are independent from one another. Therefore, for every strategy $\boldsymbol{\sigma} \in \Sigma_A^{\mathsf{pp}}$,

$$\mathbb{E}_{\boldsymbol{\lambda}}\left[\mathcal{U}'_{\boldsymbol{\lambda}}(\boldsymbol{\sigma})\right] = V_0 + \mathbb{E}_{\boldsymbol{\lambda}}\left[\sum_{i=1}^{C} \mathbb{I}[\lambda_i \leq p_i(\boldsymbol{\lambda}_{<i})] V_i(\boldsymbol{\lambda}_{<i})\right]$$

$$
\begin{aligned}
&= V_0 + \mathbb{E}_{\lambda_1} \cdots \mathbb{E}_{\lambda_C} \left[\sum_{i=1}^{C} \mathbb{I}\left[\lambda_i \le p_i(\boldsymbol{\lambda}_{<i})\right] V_i(\boldsymbol{\lambda}_{<i}) \right] && \text{(by independence)} \\
&= V_0 + \sum_{i=1}^{C} \mathbb{E}_{\boldsymbol{\lambda}_{<i}} \left[p_i(\boldsymbol{\lambda}_{<i}) V_i(\boldsymbol{\lambda}_{<i}) \right] \\
&\le (1 - p_0)\, V_0 + A. && \text{(by definition of } \Sigma_A^{\mathsf{pp}}\text{)}
\end{aligned}
$$

as claimed.

Part (c). Consider the following two strategies consisting of a single probabilistic payment after $\widetilde{\mathsf{pay}}$ (of value V_0):

- Choose $\boldsymbol{\sigma}$ such that $C := 1$, $p_1 := 1$, and $V_1 := W - V_0$. Note that $\boldsymbol{\sigma} \in \Sigma_W^{\mathsf{mp}}$ and, for every randomness choice $\boldsymbol{\lambda} \in [0,1]^C$, it holds that $\mathcal{U}'_{\boldsymbol{\lambda}}(\boldsymbol{\sigma}) = W$.
- Choose $\boldsymbol{\sigma}$ such that $C := 1$, $p_1 := 1$, and $V_1 := A - p_0 V_0$. Note that $\boldsymbol{\sigma} \in \Sigma_A^{\mathsf{pp}}$ and $\mathbb{E}_{\boldsymbol{\lambda}}\left[\mathcal{U}'_{\boldsymbol{\lambda}}(\boldsymbol{\sigma})\right] = (1 - p_0)\, V_0 + A$.

In sum, the first strategy shows that our worst-case bound is tight, while the second strategy shows that our average-case bound is tight. □

Remark 3.5 (detectable nullpayment double spends)**.** So far our analysis assumes that macropayment double spends are detectable, but nullpayment double spends are not. What if some nullpayment double spends *are* detectable? For example, merchants could maintain a partial order of all payments via a synchronous clock that ticks every second, even if the broadcast time is 10 seconds; this partial order would give chronological information on some nullpayment vs. macropayment pairs. But does such a stronger detection guarantee improve the economic bounds?

Our analysis *does* extend to this setting, and the answer is yes, but not by much. First, if some nullpayment double spends are also detectable, the additional utility of a malicious customer can only go down, so the upper bounds of our theorem continue to hold. However, the upper bounds are not tight; nevertheless, below we sketch modifications to our analysis that do recover a tight result.

- *Starting point:* the first time a detectable double spend occurs, i.e., a macropayment *or detectable nullpayment* occurs after another macropayment on the same ticket.
- *Ending point:* every merchant has detected that double spend.
- *Additional utility:* if the starting point is a macropayment double spend, the additional utility is the same, but if the starting point is a detectable nullpayment double spend, the additional utility goes down by V_0.

The rest of the analysis follows, for parameters A and W that are now defined for this new time interval. The only difference is in the initial cost of detection, due to different detection guarantees. Afterwards, only macropayment double spends provide additional utility, which are detectable in both settings. Overall, even if we had the stronger guarantee of detecting *all* nullpayment double spends, it would only save V_0 in the average-case bound.

3.3 Interpreting the payment value rates

Our analysis in Section 3.2 can be viewed as a reduction from the required deposit amount to certain per-deposit payment value rates: A (for the average case analysis), which is the maximum cumulative value of probabilistic payments across merchants within any detection time period; or W (for the worst case analysis), which is the maximum cumulative value of macropayments across merchants within the same period. Our analysis is *tight*, so leaving these parameters unbounded enables a malicious customer to gain unbounded additional utility (and rules out the possibility of deterring malicious behavior via economic means such as advance deposits). The purpose of this section is to discuss the meaning of bounding payment value rates, and what are the implications of such bounds. Throughout, recall that our analysis is *per deposit*, so we fix a single deposit $\mathbf{d}$ that backs all the probabilistic payments discussed below.

Interpretation of the parameters. We first discuss the detection time (used to define the rate), and then discuss how W and A may arise as a sum, across all merchants, of corresponding payment value rates.

- *Detection time.* We denote by T_{M} the *time for a merchant* M *to detect a detectable double spend*. For example, T_{M} can be the network's broadcast time, that is, the time for a message sent by a merchant to reach all other merchants

(this is true, e.g., if the network contains enough honest nodes to provide reliable and timely broadcast, or if merchants have the same view of the ledger). In a Bitcoin-like system the broadcast time is much smaller than the validation time (the time for a broadcast transaction to appear in the ledger): a few seconds as opposed to a few minutes.

- *Merchants (per deposit).* We denote by N the *number of merchants* that accept probabilistic payments (backed by the deposit $\mathbf{d}$). For example, N could be the number of all merchants. (Though this need not be the case, see below.)

- *Payment value rates (per deposit).* For every merchant M, $\mathfrak{a}_{\mathsf{M}} := \max_t \mathfrak{a}_{\mathsf{M}}(t)$ is the maximum (cumulative) value of probabilistic payments (backed by the deposit $\mathbf{d}$) accepted by M within any time period of T_{M}; similarly, $\mathfrak{w}_{\mathsf{M}} := \max_t \mathfrak{w}_{\mathsf{M}}(t)$ is the maximum (cumulative) value of macropayments accepted by M within any time period of T_{M}. Then one sets A equal to $\sum_{\mathsf{M}} \mathfrak{a}_{\mathsf{M}}$, and W equal to $\sum_{\mathsf{M}} \mathfrak{w}_{\mathsf{M}}$ (or consider these as upper bounds to A and W).

Necessity of bounds. We now explain why simultaneous bounds on the aforementioned parameters are necessary. First, if there is no bound on the number N of merchants that accept probabilistic payments backed by $\mathbf{d}$, a malicious customer can use $\mathbf{d}$ to gain unbounded additional utility via a one-ticket-multiple-merchant attack (see Section 3.1), even in the average case. Second, even if N is bounded (and greater than 1) but $\max_{\mathsf{M}} T_{\mathsf{M}}$ is unbounded (e.g., a large-scale eclipse attack is underway [HKZG15]), a malicious customer can gain unbounded additional utility via a multiple-ticket-multiple-merchant attack (see Section 3.1), even in the average case. Third, even if N and $\max_{\mathsf{M}} T_{\mathsf{M}}$ are bounded but some $\mathfrak{a}_{\mathsf{M}}$ is unbounded, our analysis implies that a malicious customer can again gain unbounded additional utility in the average case; similarly, if some $\mathfrak{w}_{\mathsf{M}}$ is unbounded, our analysis implies that a malicious customer can again gain unbounded additional utility in the worst case. In sum, if either $\max_{\mathsf{M}} T_{\mathsf{M}}$ or N are unbounded, then $A = \sum_{\mathsf{M}} \mathfrak{a}_{\mathsf{M}}$ and $W = \sum_{\mathsf{M}} \mathfrak{w}_{\mathsf{M}}$ are also unbounded; but even if $\max_{\mathsf{M}} T_{\mathsf{M}}$ and N are bounded, either A or W could still be unbounded, and so we must explicitly bound A or W (depending if we target average or worst case, or both).

Finally, observe that the above discussion assumes that there is no a-priori bound on how many tickets a single deposit can back; see Remark 3.6 below for a discussion of what happens if a deposit is restricted to only back macropayments up to a certain maximum total value.

Respecting the bounds. Whose responsibility is it to ensure that the bounds A or W are respected? One answer to this question could be that there are exogenous reasons (e.g., spending patterns, network behavior, and so on) that justify this statement. Another answer to this question is to say that every merchant M is responsible "for his own share": he needs to monitor that $\mathfrak{a}_{\mathsf{M}}$ and $\mathfrak{w}_{\mathsf{M}}$ are locally respected for him (and if they are about to be exceeded, he defers further payments to the next period of time T_{M}). This second answer raises an interesting technical problem: how does M know which payments are backed by the same deposit? If a payment's deposit is not private (as in [PS16]) this is not a problem. But if a payment's deposit is private, this could be tricky. In our DAM scheme construction, when engaging in a probabilistic payment, a merchant does not learn any information about the deposit that backs it, beyond the bit of whether the deposit is valid or not. Nevertheless, we still enable a merchant to get around this problem, by leveraging the notion of a *rate limit tag* within a probabilistic payment; see Section 7.2.3 for details.

Implications: good news and bad news. The good news about our economic analysis is that it gives a tight characterization of the additional utility that can be gained via double spending. The bad news is that bounding A or W may impact usability. (Perhaps this is not surprising because offline probabilistic payments are a "tough" setting since double spending cannot be fully prevented.) Namely, if all $\mathfrak{a}_{\mathsf{M}}$ (resp., $\mathfrak{w}_{\mathsf{M}}$) are large, then A (resp., W) is even larger; but this impacts usability because the required deposit is large. Conversely, if many $\mathfrak{a}_{\mathsf{M}}$ (resp., $\mathfrak{w}_{\mathsf{M}}$) are small then A (resp., W), and thus the required deposit, is not as large; but the amount of value transacted with many merchants is limited, and this impacts usability because a user may not be able to transact large amounts with his "favorite" merchants.

Mitigations. A way to mitigate the above problem is to associate to each deposit a subset $\mathcal{R}$ of allowed "receiver merchants" so that the sum is taken only over this subset: $A = \sum_{\mathsf{M} \in \mathcal{R}} \mathfrak{a}_{\mathsf{M}}$ and $W = \sum_{\mathsf{M} \in \mathcal{R}} \mathfrak{w}_{\mathsf{M}}$. Then, any particular user would only have to cover his spending habits with one (or more) deposits that cover one (or more) not-too-large subsets of merchants. The subset $\mathcal{R}$ can even be private and chosen by the user; in fact, we take this approach both when defining and constructing a DAM scheme (see Section 6 and Section 7).

Another way to mitigate the above problem is for merchants to group together into *micropayment agencies*. Such an agency acts as a proxy to the subset of merchants it serves, and its only task is to "monitor" the cumulative values of $\mathfrak{a}_{\mathsf{M}}$ and $\mathfrak{w}_{\mathsf{M}}$ for merchants in the agency. This approach does not affect any privacy guarantees from the perspective of the customer (since every probabilistic payment is anonymous from the perspective of a single merchant or any coalition of

merchants). In the extreme, one could even think of a *single* micropayment agency, and the only obstacle would be coordinating and keeping track of A and W across the network.

Remark 3.6 (bounded macropayments per deposit)**.** So far we have assumed that there is no a-priori bound on how many tickets a single deposit can back. Suppose instead that a deposit $\mathbf{d}$ can only back tickets with total macropayment value up to V_{tot}. To analyze this other setting, we can reuse ideas from our economic analysis: again, one can define additional utility by comparing the utilities of a malicious merchant and a corresponding honest merchant. We omit the analysis and simply state that the additional utility is bounded by $(2N-1)V_{\mathsf{tot}}$, where N is the number of merchants that accept probabilistic payments backed by $\mathbf{d}$ (note that in this case $\max_{\mathsf{M}} T_{\mathsf{M}}$, A, W could all be unbounded). Moreover, the bound is tight; intuitively, the maximum additional utility is achieved via a multiple-ticket-multiple-merchant attack until two macropayments with the same ticket occur for each of the N merchants.

4 Efficient fractional message transfer

A key ingredient in our construction of a DAM scheme is *fractional message transfer* (FMT): a primitive that enables probabilistic message transmission between two parties, called the 'sender' and the 'receiver'. Informally, the receiver samples a one-time key pair based on a transfer probability p; then, the sender uses the receiver's public key to encrypt a message m into a ciphertext c; finally, the receiver uses the secret key to decrypt c, thereby learning m, but only with the pre-defined probability p (and learns no information about m with probability $1 - p$). Our definition and construction of FMT are closely related to *non-interactive fractional oblivious transfer* (NFOT), which was studied in the context of 'translucent cryptography' as an alternative to key escrow [BM89, BR99]; see Section 1.1.3 for a discussion.

In this work we formulate the notion of an *FMT scheme*, which formally captures the functionality and security of probabilistic message transmission; we rely on this tool (and others) in our construction of a DAM scheme in Section 7. Moreover, we give an efficient construction of an FMT scheme that works for transfer probabilities $p = 1/n$ with $n \in \mathbb{N}$; this construction is in the random oracle model and assumes the hardness of the DDH problem in prime-order groups. Finally, since probabilistic message transmission is of independent interest, we also define the notion of an *FMT protocol* via an ideal functionality, and show that the security definition of FMT schemes does imply security relative to that ideal functionality. (Our DAM scheme relies on an FMT scheme, rather than an FMT protocol, because we interleave the FMT scheme with other building blocks.)

We defer the definitions, constructions, and proofs about FMT to Appendix A. In the rest of this section, we informally describe the syntax, correctness, and security of FMT schemes, and then sketch our FMT construction.

Syntax. An FMT scheme is a quintuple of algorithms $(\mathsf{FMT.Setup}, \mathsf{FMT.Keygen}, \mathsf{FMT.Encrypt}, \mathsf{FMT.Decrypt})$ with the following syntax.

- *Parameter setup (executed by a trusted party):* $\mathsf{FMT.Setup}(1^\lambda) \to \mathsf{pp_{FMT}}$. On input a security parameter λ, $\mathsf{FMT.Setup}$ outputs the public parameters $\mathsf{pp_{FMT}}$ for the scheme.
- *Key generation (executed by the receiver):* $\mathsf{FMT.Keygen}(\mathsf{pp_{FMT}}, p) \to (\mathsf{pk_{FMT}}, \mathsf{sk_{FMT}})$. On input public parameters $\mathsf{pp_{FMT}}$ and a transfer probability p, $\mathsf{FMT.Keygen}$ outputs a one-time key pair $(\mathsf{pk_{FMT}}, \mathsf{sk_{FMT}})$.
- *Message encryption (executed by the sender):* $\mathsf{FMT.Encrypt}(\mathsf{pp_{FMT}}, \mathsf{pk_{FMT}}, m) \to c$. On input public parameters $\mathsf{pp_{FMT}}$, a public key $\mathsf{pk_{FMT}}$ and a message m, $\mathsf{FMT.Encrypt}$ outputs a ciphertext c.
- *Message decryption (executed by the receiver):* $\mathsf{FMT.Decrypt}(\mathsf{pp_{FMT}}, \mathsf{sk_{FMT}}, c) \to m'$. On input public parameters $\mathsf{pp_{FMT}}$, a secret key $\mathsf{sk_{FMT}}$ and a ciphertext c, $\mathsf{FMT.Decrypt}$ outputs a message m' that equals m or $\varnothing$. (The special symbol $\varnothing$ denotes that decryption resulted in no message.)

An FMT scheme satisfies the correctness and security properties defined below.

Correctness. An FMT scheme is *correct* if for every security parameter λ, public parameters $\mathsf{pp_{FMT}} \in \mathsf{FMT.Setup}(1^\lambda)$, transfer probability $p \in \mathcal{P} \subseteq [0, 1]$, key pair $(\mathsf{pk_{FMT}}, \mathsf{sk_{FMT}}) \in \mathsf{FMT.Keygen}(\mathsf{pp_{FMT}}, p)$, and message $m \in \mathcal{M}$,

$$\mathsf{FMT.Decrypt}(\mathsf{pp_{FMT}}, \mathsf{sk_{FMT}}, \mathsf{FMT.Encrypt}(\mathsf{pp_{FMT}}, \mathsf{pk_{FMT}}, m)) = \begin{cases} m & \text{w.p.} \quad p \\ \varnothing & \text{w.p.} \quad 1 - p \end{cases}$$

where the probability is taken over the randomness of $\mathsf{FMT.Encrypt}$ (and $\mathsf{FMT.Decrypt}$ is deterministic).

Security. An FMT scheme is *secure* if it has the properties of *fractional hiding* and *fractional binding*. Informally, fractional hiding says that an honest encryptor transferring a message m can be sure that the decryptor, who knows the secret key, learns m with probability exactly p (and $\varnothing$ with probability $1 - p$), even if the public key was generated maliciously. Fractional binding says that, for every $p' \neq p$, a malicious encryptor cannot produce a valid ciphertext that decrypts with probability p' to a valid message (i.e., not $\varnothing$).

An efficient FMT scheme. Our construction of an FMT scheme targets the case where p equals $1/n$ for some positive integer n; this case suffices within our construction of a DAP scheme. As in prior work [BM89, BR99], our starting point is the Elgamal encryption scheme [Elg85], whose semantic security relies on the hardness of DDH in prime-order groups. We now give an informal sketch of our construction.

- $\mathsf{FMT.Setup}(1^\lambda)$: sample a group $\mathbb{G}$ of prime order q (depending on λ), along with two generators $g, g_0 \in \mathbb{G}$.
- $\mathsf{FMT.Keygen}(\mathsf{pp_{FMT}}, p)$: the public key contains a Pedersen commitment [Ped91] to a random s in $\{1, \ldots, n\}$ and the secret key contains the commitment's randomness; that is, the commitment is $h = g_0^{-s} g^\alpha$ for random $\alpha \in \mathbb{Z}_q$.
- $\mathsf{FMT.Encrypt}(\mathsf{pp_{FMT}}, \mathsf{pk_{FMT}}, m)$: sample random $r \in \mathbb{Z}_q$ and random $t \in \{1, \ldots, n\}$, and use h as an Elgamal public key to encrypt the message $m' := m \cdot g_0^{rt}$; the resulting ciphertext is $c = (t, c_1, c_2) = (t, g^r, m' h^r)$.
- $\mathsf{FMT.Decrypt}(\mathsf{pp_{FMT}}, \mathsf{sk_{FMT}}, c)$: use the secret key α to decrypt the ciphertext by setting $m'' := c_2 / c_1^\alpha = m g_0^{r(t-s)}$.

The above sketch omits several important details. In particular, our construction also includes NIZKs (obtained via the Fiat–Shamir transform applied to simple Σ-protocols) to prove correctness of key generation and encryption. Informally, our FMT's correctness and security follow from the fact that $m'' = m$ only when $t = s$, which occurs with probability $p = 1/n$. The full construction and proof of security (based on hardness of DDH) are in Appendix A.

Remark 4.1 (one-time key pairs)**.** The key pair generated by the receiver *can* be reused *only if* the receiver does not leak information about whether the ciphertext decrypted to a message or $\varnothing$. As in prior constructions [BM89, BR99], this information helps senders evade fractional hiding and fractional binding, and therefore it is best to think of the key pair as suitable for only a single encryption (or a limited number of encryptions).

Remark 4.2 (security of FMT)**.** The security of prior work on constructing FMT schemes relied on the CDH assumption [BM89, BR99]. This is because this work only needed to hide information about random messages. Since we use FMT schemes to hide non-random messages, the security of our construction relies on the DDH assumption, which results in a semantic-security-like property for the hidden message.

Furthermore, our DAM scheme construction allows the same party to play the role of both sender and receiver. Thus, to prove security, we require the FMT scheme to be simultaneously simulatable and extractable in the same experiment. Our construction achieves this property via our usage of simulation-extractable NIZKs [Sah99, DDO+01].

5 Recalling decentralized anonymous payments

We review the interface of a *decentralized anonymous payment* (DAP) scheme, the primary building block in our construction of a DAM scheme (see Section 7); for details on their construction and security, see [BCG+14]. In this paper, we use DAP schemes that generalize in a straightforward way the ones in [BCG+14, GGM16] (see Section 5.4 for a comparison), and also relies on a stronger simulation-based security definition, discussed in Appendix B.

Informally, a DAP scheme is a cryptographic primitive that enables parties to conduct privacy-preserving non-interactive payments to one another. Recall that DAP schemes are formulated in a model where all parties have access to an ideal append-only log, which we call *ledger* and denote by $\mathbf{L}$.[7] We treat $\mathbf{L}$ as an *oracle*; for instance, when an algorithm needs to read the ledger, we specify $\mathbf{L}$ in the algorithm's superscript. Each entry in the ledger is called a *transaction*, denoted tx; anyone can append a new transaction to the ledger by invoking $\mathbf{L}.\mathsf{Push}(\mathsf{tx})$. The current number of entries in the ledger is $\mathbf{L}.\mathsf{Len}$. The ledger is additionally partitioned into *epochs*, and we denote the current epoch by $\mathbf{L}.\mathsf{Epoch}$.Epochs are incremented with the agreement of all participating parties; this consensus is achieved when every party has invoked $\mathbf{L}.\mathsf{IncrementEpoch}$ during the current epoch.

5.1 Data structures

The main data structures in DAP schemes are *coins*, which store funds, and *transactions*, which create or transfer funds.

Coins. A *coin*, denoted by the symbol $\mathbf{c}$, is a data structure that stores funds. A coin $\mathbf{c}$ has the following attributes: (a) a *commitment* cm, which computationally binds together all other attributes of $\mathbf{c}$ while hiding any statistical information about them; (b) an *address public key* apk, which represents $\mathbf{c}$'s owner; (c) a *value* v, which represents $\mathbf{c}$'s monetary value; (d) an arbitrary *public information string* pub; and (e) an arbitrary *secret information string* sec.

Spending a coin $\mathbf{c}$ requires knowing the *address secret key* ask associated with $\mathbf{c}$'s address public key apk. Spending $\mathbf{c}$ reveals the coin's *serial number* sn, which is deterministically derived from $\mathbf{c}$ and ask (but is pseudorandom without knowing ask); we denote by GetSN this computation, so that $\mathsf{sn} = \mathsf{GetSN}(\mathbf{c}, \mathsf{ask})$. It is computationally infeasible to create two distinct coins that share the same serial number sn but have differing secret information sec and sec'. The coin $\mathbf{c}$ is also associated with a *coin identifier* id, which is a string that identifies $\mathbf{c}$ uniquely (with all but negligible probability) and is deterministically derived from $\mathbf{c}$ and ask; we denote by GetID this computation, so that $\mathsf{id} = \mathsf{GetID}(\mathbf{c}, \mathsf{ask})$. Note that id and sn are *different*; in particular, id is *not* revealed when $\mathbf{c}$ is spent. Furthermore, it is computationally infeasible to create two coins that have differing serial numbers but the same identifier.

Transactions. A *transaction*, denoted by the symbol tx, is a data structure that creates or transfers funds. Transactions are created by users and then appended to the ledger $\mathbf{L}$ for anyone to verify; despite being public, they are privacy-preserving as described below. There are two types of transactions.

- *Mint transaction, denoted* tx_m. A mint transaction records the creation of a new coin $\mathbf{c}$. The transaction equals the tuple $(\mathsf{cm}, v, \mathsf{pub}, *)$, where cm is $\mathbf{c}$'s commitment, v is $\mathbf{c}$'s value, pub is $\mathbf{c}$'s public information, and $*$ is other implementation-dependent information. While anyone can verify that v and pub are bound to cm, the transaction does not leak any other information about $\mathbf{c}$.

- *Pour transaction, denoted* tx_p. A pour transaction records the transfer of funds from 'old' coins to 'new' coins (possibly owned by other parties). The transaction equals the tuple $(\mathsf{ts}, \Delta, [\mathsf{sn}_i]_1^m, [\mathsf{cm}_j]_1^n, v_\mathsf{pub}, \mathsf{info}, *)$, where ts is the timestamp (recording the epoch of when this transaction was created), Δ is the activation delay (discussed below), $[\mathsf{sn}_i]_1^m$ are the serial numbers of the old coins, $[\mathsf{cm}_j]_1^n$ are the commitments of the new coins, v_pub is part of the value of the old coins that is revealed for public payments (e.g., for transaction fees), info is an arbitrary information string associated with the transaction, and $*$ is other implementation-dependent information.

  A pour transaction reveals no information about the values of old coins or new coins, besides: (i) the fact that the sum of the old coin's values is at least v_pub, and that the rest is distributed among the new coins; (ii) the serial numbers of old coins (thereby 'consuming' the old coins); and (iii) the commitments of the new coins.

  The activation delay Δ specifies that tx_p becomes *active* Δ epochs after when it is pushed to the ledger (Δ can be 0); until then, tx_p remains *dormant*. See Section 5.3 for how the ledger handles this aspect.

[7] In reality, protocols for realizing the ledger, such as that of [Nak09], are not perfect and are subject to temporary inconsistencies, re-orderings of transactions in flight from parties to the ledger, and so on; in this respect, DAP schemes are only as good as the underlying ledger.

5.2 Algorithms

A DAP scheme is a tuple of algorithms (some of which take the ledger $\mathbf{L}$ as oracle):

$$\mathsf{DAP} = (\mathsf{Setup}, \mathsf{CreateAddr}, \mathsf{Receive}^{\mathbf{L}}, \mathsf{Mint}^{\mathbf{L}}, \mathsf{Pour}^{\mathbf{L}}, \mathsf{VerifyTransaction}^{\mathbf{L}}) \ .$$

The syntax and semantics of these algorithms are informally described below; when necessary, an algorithm has oracle access to the ledger $\mathbf{L}$. For our convenience, we extend/modify [BCG+14]'s interface of a DAP scheme; see Section 5.4 below for a comparison.

System setup: $\mathsf{DAP.Setup}(1^\lambda, \Pi_\mathsf{m}, \Pi_\mathsf{p}) \to \mathsf{pp}$

On input a security parameter 1^λ, a *mint predicate* Π_m, and a *pour predicate* Π_p, DAP.Setup outputs public parameters pp for the system. The predicates $\Pi_\mathsf{m}, \Pi_\mathsf{p}$ are included in pp, and thus are public; Π_m 'regulates' which mints of new coins are legal, and Π_p does the same for pours of old coins into new ones (see below for more detail). A trusted party runs this algorithm once and then publishes the public parameters; afterwards the trusted party is not needed anymore.

Creating addresses: $\mathsf{DAP.CreateAddr}(\mathsf{pp}, \mathsf{meta}) \to (\mathsf{apk}, \mathsf{ask})$

On input public parameters pp and address metadata meta, DAP.CreateAddr outputs an address key pair $(\mathsf{apk}, \mathsf{ask})$. The address metadata is bound to apk in the sense that it is computationally infeasible to create two address public keys that are equal in all aspects except in differing metadata. Any user may run this algorithm to create an address key pair; the address public key is to receive coins while the address secret key is to send coins.

Receiving coins: $\mathsf{DAP.Receive}^{\mathbf{L}}(\mathsf{pp}, (\mathsf{apk}, \mathsf{ask})) \to$ set of (unspent) received coins

On input public parameters pp and an address key pair $(\mathsf{apk}, \mathsf{ask})$, and given oracle access to the ledger $\mathbf{L}$, DAP.Receive outputs the set of unspent coins sent to the address public key apk. (To retrieve unspent coins from a particular pour transaction, one can run DAP.Receive on the "ledger" consisting of just that one transaction.)

Minting coins: The process of creating new coins is called *minting*, and its inputs and outputs are as follows.

$\mathsf{DAP.Mint}^{\mathbf{L}}$
public parameters pp value v address public key apk public information string pub secret information string sec
new coin $\mathbf{c}$ mint transaction tx_m

The output coin $\mathbf{c}$ has value v, address apk, public information pub, and secret information sec; the coin $\mathbf{c}$ is to be kept secret for later spending. A coin's attribute choices are regulated by the mint predicate Π_m: minting succeeds if and only if $\Pi_\mathsf{m}(\mathbf{L}.\mathsf{Epoch}, v, \mathsf{apk}, \mathsf{pub}, \mathsf{sec}) = 1$. The output mint transaction tx_m is to be published by invoking $\mathbf{L}.\mathsf{Push}(\mathsf{tx}_\mathsf{m})$.

Pouring coins: The process of transferring value from 'old' input coins to 'new' output coins is called *pouring*; the old coins are consumed while the new coins are minted; a part v_pub of the input value can be publicly revealed (e.g., to pay a transaction fee). Pouring thus allows users to subdivide coins into smaller denominations, merge coins, transfer ownership of coins, or make public payments — up to restrictions imposed by the pour predicate Π_p (see below). The algorithm that realizes this is called DAP.Pour, and its inputs and outputs are as follows.

$\mathsf{DAP.Pour}^{\mathbf{L}}$
public parameters pp old coins $[\mathbf{c}_i]_1^m$ old address secret keys $[\mathsf{ask}_i]_1^m$ new values $[v_j]_1^n$ new address public keys $[\mathsf{apk}_j]_1^n$ new public information strings $[\mathsf{pub}_j]_1^n$ new secret information strings $[\mathsf{sec}_j]_1^n$ public value v_{pub} transaction information string info activation delay Δ
new coins $[\mathbf{c}_j]_1^n$ pour transaction $\mathsf{tx_p}$

In other words, DAP.Pour takes as input a list of old coins $[\mathbf{c}_i]_1^m$, along with corresponding secret keys $[\mathsf{ask}_i]_1^m$, attributes for new coins analogous to DAP.Mint's inputs, a public value v_{pub}, an arbitrary transaction information string info, and an activation delay Δ (discussed below). The outputs of DAP.Pour consist of the new coins $[\mathbf{c}_j]_1^n$ and a pour transaction $\mathsf{tx_p}$ (whose timestamp ts equals $\mathbf{L}$.Epoch and activation delay is Δ). The output coins $[\mathbf{c}_j]_1^n$ are to be kept secret for later use, while the output pour transaction $\mathsf{tx_p}$ is to be published by invoking $\mathbf{L}.\mathsf{Push}(\mathsf{tx_p})$.

Analogously to minting, attribute choices while pouring are regulated by the pour predicate Π_{p}: pouring succeeds if and only if $\Pi_{\mathsf{p}}(\mathbf{L}.\mathsf{Epoch}, [\mathbf{c}_i]_1^m, [\mathsf{ask}_i]_1^m, [v_j]_1^n, [\mathsf{apk}_j]_1^n, [\mathsf{pub}_j]_1^n, [\mathsf{sec}_j]_1^n, v_{\mathsf{pub}}, \mathsf{info}, \Delta) = 1$.

Finally, the activation delay Δ specifies that $\mathsf{tx_p}$ becomes active Δ epochs after when $\mathsf{tx_p}$ is added to the ledger; see Section 5.3. The epoch when $\mathsf{tx_p}$ is added to the ledger is never less than $\mathsf{tx_p}$'s timestamp but it could be greater than it.

Verifying transactions: $\mathsf{DAP.VerifyTransaction}^{\mathbf{L}}(\mathsf{pp}, \mathsf{tx}) \to b$

On input public parameters pp and a (mint or pour) transaction tx, and given oracle access to the ledger $\mathbf{L}$, DAP.VerifyTransaction outputs a bit b denoting whether the transaction tx is valid relative to the ledger $\mathbf{L}$.

5.3 Merkle trees on all coin commitments and all serial numbers

For efficiency reasons, a DAP scheme relies on two data structures publicly maintained alongside the ledger $\mathbf{L}$: (1) a Merkle tree $\mathbf{T}_{\mathrm{cm}}$ over all coin commitments in $\mathbf{L}$, taken in any order; (2) a Merkle tree $\mathbf{T}_{\mathrm{sn}}$ over all serial numbers in $\mathbf{L}$, *taken in lexicographic order*. These are updated as new transactions are pushed to the ledger, as follows.

Whenever a mint transaction $\mathsf{tx_m}$ is pushed to the ledger, the coin commitment in $\mathsf{tx_m}$ is added to $\mathbf{T}_{\mathrm{cm}}$. Whenever a pour transaction $\mathsf{tx_p}$ is pushed to the ledger, if $\mathsf{tx_p}$'s activation delay Δ equals zero, then $\mathsf{tx_p}$ immediately becomes active: its coin commitments and serial numbers are added to $\mathbf{T}_{\mathrm{cm}}$ and $\mathbf{T}_{\mathrm{sn}}$. However, if Δ is positive, then $\mathsf{tx_p}$ becomes active Δ epochs later, so that $\mathsf{tx_p}$'s coin commitments and serial numbers are *not* immediately added to the two trees. Instead, this happens Δ epochs later *only if none of the* Δ *transactions in this period is another pour transaction containing any of* $\mathsf{tx_p}$*'s serial numbers* (if such a pour transaction does exist then $\mathsf{tx_p}$ remains dormant forever). In summary, the Merkle trees $\mathbf{T}_{\mathrm{cm}}$ and $\mathbf{T}_{\mathrm{sn}}$ consider only commitments and serial numbers in active transactions, and ignore dormant ones.

Finally, note that a transaction can be pushed to the ledger only if any commitments or serial numbers that it contains do not already appear in the respective Merkle trees (i.e., duplicate commitments or serial numbers are forbidden).

Additional commitments and serial numbers. Additional commitments or serial numbers of coins may be revealed, due to application-dependent reasons, in the '$*$' field of a mint/pour transaction, or even in new types of transactions. The trees $\mathbf{T}_{\mathrm{cm}}$ and $\mathbf{T}_{\mathrm{sn}}$ are over *all* (active) commitments and serial numbers appearing *anywhere* on the ledger $\mathbf{L}$.

Notation for $\mathbf{T_{cm}}$ & $\mathbf{T_{sn}}$. Later on we prove membership of coin commitments in $\mathbf{T}_{\mathrm{cm}}$, and so for notational convenience we make explicit the following functionality: (1) $\mathbf{L}.\mathsf{CMRoot}$ returns the current root of $\mathbf{T}_{\mathrm{cm}}$; (2) $\mathbf{L}.\mathsf{CMProve}(\mathsf{cm})$ returns a proof π_{cm} of the fact that cm is in $\mathbf{T}_{\mathrm{cm}}$; (3) $\mathbf{L}.\mathsf{CMVerify}(\mathsf{rt}, \mathsf{cm}, \pi_{\mathsf{cm}})$ outputs 1 if and only if π_{cm} is a valid proof of membership of cm when $\mathbf{T}_{\mathrm{cm}}$ had root rt. We also prove membership *and non-membership* of serial numbers in $\mathbf{T}_{\mathrm{sn}}$, and so we make explicit the following functionality: (1) $\mathbf{L}.\mathsf{SNRoot}$ returns the current root of $\mathbf{T}_{\mathrm{sn}}$; (2) $\mathbf{L}.\mathsf{SNProve}(\mathsf{sn})$ returns a bit b_{sn} indicating whether sn is in $\mathbf{T}_{\mathrm{sn}}$, and a proof π_{sn} of this fact; (3) $\mathbf{L}.\mathsf{SNVerify}(\mathsf{rt}, \mathsf{sn}, b_{\mathsf{sn}}, \pi_{\mathsf{sn}})$ outputs 1 if and only if π_{sn} is a valid proof of membership ($b_{\mathsf{sn}} = 1$) or of non-membership ($b_{\mathsf{sn}} = 0$) of sn when $\mathbf{T}_{\mathrm{sn}}$ had root rt. Note that membership and non-membership in sorted Merkle trees can be proved efficiently [MRK03].

5.4 Extension of the DAP interface

The DAP scheme interface described above generalizes the ones in [BCG+14] and [GGM16]. These straightforward extensions simplify exposition when constructing a DAM scheme in Section 7. Below we summarize these differences:

- **Any arity.** DAP.Pour is no longer restricted to only two input and output coins; it now takes an arbitrary number of input coins and outputs an arbitrary (and possibly different) number of output coins. This 'arity' information is not private because the number of input coins and the number of output coins are revealed in a pour transaction.

- **Ledger as oracle.** DAP.Pour no longer takes a Merkle tree root and authentication paths as inputs but, instead, obtains these via oracle access to the ledger $\mathbf{L}$. Similarly, DAP.VerifyTransaction and DAP.Receive no longer take the ledger $\mathbf{L}$ as input but instead access it as an oracle.

- **New attribute of an address.** An address key pair $(\mathsf{apk}, \mathsf{ask})$ has an additional attribute: its metadata meta. Recall that [BCG+14]'s construction generates address public keys as follows: sample ask and then set $\mathsf{apk} := \mathsf{PRF}_{\mathsf{ask}}(0)$. This can be modified to support the address metadata attribute by evaluating the PRF at meta instead of at 0, and then accompanying the address with a zero knowledge proof of knowledge that this step was performed correctly.

- **New attributes of a coin.** A coin $\mathbf{c}$ has additional attributes. Note that $\mathbf{c}$'s commitment cm computationally binds together all (previous and additional) attributes of $\mathbf{c}$ while hiding any statistical information about them.

  – *Identifier.* A coin $\mathbf{c}$ is associated with a random string id (sampled at minting) that identifies $\mathbf{c}$ uniquely (with all but negligible probability). Note that $\mathbf{c}$'s identifier id *is different from* $\mathbf{c}$'s serial number sn in that it is not used for prevention of double spending, and is not revealed when the $\mathbf{c}$ is poured to another coin. Moreover, it is difficult to create two coins that have differing serial numbers but the same identifier.

  – *Public and secret information.* A coin $\mathbf{c}$ is associated with two additional pieces of information: a public information string pub and a secret information string sec. In a mint transaction, pub is publicly revealed but no information about sec is revealed; in a pour transaction, no information about pub or sec is revealed (beyond what is implied by the transaction's validity). We assume that it is computationally infeasible to create coins that have the same serial number sn but differing secret information sec (in [BCG+14]'s construction, it suffices to append sec to the serial number randomness). To incorporate these changes, DAP.Mint additionally takes both pub and sec as input, and DAP.Pour additionally takes analogous parameters for specifying the new coins.

- **Mint and pour predicates.** DAP.Setup takes two additional inputs: a mint predicate Π_{m} and a pour predicate Π_{p}. These are included in the public parameters pp, and thus are public. The minting of new coins is now regulated by Π_{m}, which receives the current epoch and the new coin's attributes, and accepts or rejects; that is, minting succeeds if and only if $\Pi_{\mathsf{m}}(\mathbf{L}.\mathsf{Epoch}, v, \mathsf{apk}, \mathsf{pub}, \mathsf{sec}) = 1$. Similarly, the pouring of old coins into new coins is now regulated by Π_{p}, which receives the current epoch and information about the old and new coins, and accepts or rejects; that is, pouring succeeds if and only if $\Pi_{\mathsf{p}}(\mathbf{L}.\mathsf{Epoch}, [\mathbf{c}_i]_1^m, [\mathsf{ask}_i]_1^m, [v_j]_1^n, [\mathsf{apk}_j]_1^n, [\mathsf{pub}_j]_1^n, [\mathsf{sec}_j]_1^n, v_{\mathsf{pub}}, \mathsf{info}, \Delta) = 1$.

  Via a suitable choice of Π_{m} and Π_{p}, one can overlay additional semantics on minting and pouring. For example, we use Π_{p} to realize different coin types and allow only certain type transitions. Extending [BCG+14]'s construction with this functionality is straightforward: mint transactions simply include a zero-knowledge proof that Π_{m} accepted; similarly, it suffices to include Π_{p} as a sub-circuit of the pour circuit used in DAP.Pour.

  Finally, while Π_{m} and Π_{p} only receive $\mathbf{L}.\mathsf{Epoch}$ as input, one can consider a more general setting where Π_{m} and Π_{p} have oracle access to $\mathbf{L}$, and use more complex information about $\mathbf{L}$ to make a decision. In this paper we do not need this level of generality, but this additional freedom may be attractive for other applications.

6 Definition of a decentralized anonymous micropayment scheme

We define *decentralized anonymous micropayment* (DAM) schemes, which extend DAP schemes [BCG+14] with probabilistic payments backed by advance deposits. (Our economic analysis in Section 3 discusses the appropriate value of deposits.) This extension necessitates discussing the new kinds of data structures used by a DAM scheme's interface, and also requires a new security definition that captures not only the notion of financial integrity but also (as foreshadowed in Section 2.1) strong forms of unlinkability across probabilistic payments and other properties.

We now proceed as follows: we introduce a DAM scheme's data structures (Section 6.1), algorithms (Section 6.2), usage (Section 6.3), and security (Section 6.4). Later on, we present our construction (Section 7).

6.1 Data structures

In a DAP scheme all payments are of the same type: old coins are consumed to produce new coins (perhaps belonging to a different user). A DAM scheme, however, has a richer set of payment types: coins can be consumed to produce not only new coins, but also to produce *deposits* or (lottery) *tickets*; in turn, a ticket can be consumed probabilistically: with a certain probability, it results in a coin for the receiver (and results in a nullpayment otherwise). Finally, a DAM scheme allows users to perform two kinds of operations on deposits: withdrawal, which results in coins, and punishment, which allows users to punish other cheating users for double spending (thereby invalidating their deposit).

To support the above richer semantics, a DAM scheme needs a richer set of data structures than a DAP scheme. We describe these below. To avoid ambiguity between different "coin types", we follow the terminology of Zcash [HBHW16] and refer to the different "coin" data structures as *notes*.

Notes. In a DAM scheme, funds are stored in *notes*, denoted $\mathbf{n}$. There are three types of notes.

- *Coin notes (also, coins), denoted* $\mathbf{c}$. These notes are semantically analogous to coins in a DAP scheme: one can mint coins or pour coins into new coins. In addition, one can also pour coins into either of the two note types below.
- *Deposit notes (also, deposits), denoted* $\mathbf{d}$. These notes are used to back ticket notes (see next type) and disincentivize double spending. One can mint or withdraw deposits. Withdrawals are delayed to prevent front running.
- *Ticket notes (also, tickets), denoted* $\mathbf{t}$. These notes are used to engage in a probabilistic payments: with a certain probability the ticket is poured into a coin, and with the remaining probability it remains unspent and can be used in future probabilistic payments. Each ticket is linked to a (private) deposit that backs it. Tickets can be *refreshed* to prevent loss of money or anonymity due to merchant aborts; refreshes are also delayed to prevent front running.

We denote the above note types by the symbols $\mathtt{cn}, \mathtt{dp}, \mathtt{tk}$; a note's type is stored in its public information string $\mathtt{pub}$.

Transactions. In a DAM scheme, there are six types of transactions.

- *Mint transactions, denoted* tx_{m}. A DAM mint transaction is analogous to a DAP mint transaction (see Section 5.1); it records the creation of a new note $\mathbf{n}$ (having one of the above types).
- *Pour-coin transactions, denoted* $\mathsf{tx}_{\mathsf{pc}}$. A pour-coin transaction $\mathsf{tx}_{\mathsf{pc}}$ is analogous to a DAP pour transaction (see Section 5.1); it records the transfer of funds from 'old' coins into 'new' notes (having one of the above types).
- *Pour-ticket transactions, denoted* $\mathsf{tx}_{\mathsf{pt}}$. A pour-ticket transaction records the transfer of funds from an 'old' ticket into a 'new' coin; it represents the fact that a probabilistic payment resulted in a macropayment (nullpayments do not have corresponding transactions). A pour-ticket transaction consists of a DAP pour transaction tx_{p} that stores additional information that enables verification of the macropayment.
- *Withdraw transactions, denoted* $\mathsf{tx}_{\mathsf{wd}}$. A withdraw transaction records the transfer of funds from an 'old' deposit into a 'new' coin. The transaction consists of a suitable DAP pour transaction (and inherits its privacy guarantees).
- *Refresh transactions, denoted* $\mathsf{tx}_{\mathsf{ref}}$. A refresh transaction records the transfer of funds from an 'old' ticket into a 'new' ticket. The transaction consists of a suitable DAP pour transaction (and inherits its privacy guarantees).
- *Punish transactions, denoted* $\mathsf{tx}_{\mathsf{pun}}$. A punish transaction records a detected double spend. The transaction equals the tuple $(\mathsf{tx}_{\mathsf{pt}}, \mathsf{tx}'_{\mathsf{pt}}, \mathsf{sn}, \mathsf{id})$, where the two pour-ticket transactions $\mathsf{tx}_{\mathsf{pt}}, \mathsf{tx}'_{\mathsf{pt}}$ are proof of a double-spent ticket, and sn and id are the serial number and unique identifier of the deposit $\mathbf{d}$ that backed that ticket in both macropayments.

Note that pour-coin, pour-ticket, withdraw, and refresh transactions are instances of DAP pour transactions; they differ in the types of the input and output notes, and also in their activation delay. In particular, pour-coin and pour-ticket transactions have activation delay zero, while withdraw and refresh transactions have activation delays Δ_{w} and Δ_{r},

which are predetermined system constants. These delays prevent cheating customers from evading punishment by front running honest merchants seeking to punish them.

6.2 Algorithms

A DAM scheme specifies various algorithms and one 2-party protocol; some of these take the ledger $\mathbf{L}$ as oracle.

$$\mathsf{DAM} = \left(\begin{array}{lllll} \mathsf{Setup} & \mathsf{MintCoin}^{\mathbf{L}} & \mathsf{PourCoinToCoin}^{\mathbf{L}} & & \\ \mathsf{CreateAddr} & \mathsf{MintDeposit}^{\mathbf{L}} & \mathsf{PourCoinToDeposit}^{\mathbf{L}} & \mathsf{WithdrawDeposit}^{\mathbf{L}} & \mathsf{Punish} \\ \mathsf{Receive}^{\mathbf{L}} & \mathsf{MintTicket}^{\mathbf{L}} & \mathsf{PourCoinToTicket}^{\mathbf{L}} & \mathsf{RefreshTicket}^{\mathbf{L}} & \mathsf{VerifyTransaction}^{\mathbf{L}} \\ & & \mathsf{PourTicket}^{\mathbf{L}} & & \end{array} \right) .$$

We describe the syntax and semantics of each of these below.

System setup: $\mathsf{DAM.Setup}(1^\lambda) \to \mathsf{pp}$

On input a security parameter 1^λ, DAM.Setup outputs public parameters pp for the system. A trusted party runs this algorithm once and then publishes the public parameters; afterwards the trusted party is not needed anymore.

Creating addresses: $\mathsf{DAM.CreateAddr}(\mathsf{pp}, \mathsf{spec}) \to (\mathsf{apk}, \mathsf{ask})$

On input public parameters pp and a *probabilistic payment specification* spec, DAM.CreateAddr outputs an address key pair $(\mathsf{apk}, \mathsf{ask})$. The specification $\mathsf{spec} := (\mathfrak{a}, \mathfrak{w}, p, V)$ consists of the average-case payment rate $\mathfrak{a}$, the worst-case payment rate $\mathfrak{w}$, transfer probability p, and payment value V for a probabilistic payment. (Users not intending to receive probabilistic payments to apk can set its spec to be $\perp$.) Any user may run this algorithm to create an address key pair; the address public key is to receive notes while the address secret key is to send notes.

Receiving payments: $\mathsf{DAM.Receive}^{\mathbf{L}}(\mathsf{pp}, (\mathsf{apk}, \mathsf{ask})) \to$ set of (unspent) received notes

On input public parameters pp and an address key pair $(\mathsf{apk}, \mathsf{ask})$, and given oracle access to the ledger $\mathbf{L}$, DAM.Receive outputs the set of unspent notes sent to the address public key apk. (To retrieve unspent notes from a particular transaction, one can run DAM.Receive on the "ledger" consisting of just that one transaction.)

Minting notes: The process of creating new notes is called *minting*. The minting procedure depends on whether one wants to mint a coin, a deposit, or a ticket. Hence, a DAM scheme includes a triple of algorithms

$$(\mathsf{DAM.MintCoin}^{\mathbf{L}}, \mathsf{DAM.MintDeposit}^{\mathbf{L}}, \mathsf{DAM.MintTicket}^{\mathbf{L}}) \ .$$

Each of these algorithms takes as input public parameters pp, a note value v, and an address public key apk, and outputs the minted note (whose type is stored in its public information pub) and a corresponding mint transaction $\mathsf{tx_m}$. Moreover:

- When minting a deposit, the minting procedure takes as additional input a *receiver address set* $\mathcal{R}$, and a commitment to $\mathcal{R}$ is stored in the deposit's secret information string sec; $\mathcal{R}$ specifies the addresses of merchants that can be paid with tickets backed by this deposit (see Section 3.3).
- When minting a ticket, the minting procedure takes as additional input a deposit, which is stored in the ticket's secret information sec.

The table below summarizes the inputs and outputs for these three algorithms. Note that the output mint transaction $\mathsf{tx_m}$ is to be published by invoking $\mathbf{L}.\mathsf{Push}(\mathsf{tx_m})$.

$\mathsf{DAM.MintCoin}^{\mathbf{L}}$	$\mathsf{DAM.MintDeposit}^{\mathbf{L}}$	$\mathsf{DAM.MintTicket}^{\mathbf{L}}$
public parameters pp value v address public key apk		
—	receiver address set $\mathcal{R}$	deposit number $\mathbf{d}$
coin $\mathbf{c}$ such that $\mathbf{c}.\mathtt{pub} = \mathtt{cn}$ $\mathbf{c}.\mathsf{sec} = \perp$	deposit $\mathbf{d}$ such that $\mathbf{d}.\mathtt{pub} = \mathtt{dp}$ $\mathbf{d}.\mathsf{sec} =$ a commitment to $\mathcal{R}$	ticket $\mathbf{t}$ such that $\mathbf{t}.\mathtt{pub} = \mathtt{tk}$ $\mathbf{t}.\mathsf{sec} = \mathbf{d}$
mint transaction $\mathsf{tx_m}$		

Pouring notes: The process of transferring value from notes to other notes is called *pouring*; these 'old' and 'new' notes can be of different types. The pour procedure depends on whether one wants to pour a coin or a ticket. In the first case, the procedure consists of three separate algorithms, depending on the desired new type: DAM.PourCoinToCoin, DAM.PourCoinToDeposit, and DAM.PourCoinToTicket. The three algorithms are used to pour a set of 'old' input coins into a set of 'new' output coins, deposits, or tickets respectively; the output pour-coin transaction $\mathsf{tx}_{\mathsf{pc}}$ is to be published by invoking $\mathbf{L}.\mathsf{Push}(\mathsf{tx}_{\mathsf{pc}})$.

If instead one wants to pour a ticket, then the procedure consists of a *protocol* $\mathsf{DAM.PourTicket}^{\mathbf{L}}$, which is used to run a probabilistic payment between two parties called the *sender* and *receiver*; with a certain probability, the receiver gets a pour-ticket transaction $\mathsf{tx}_{\mathsf{pt}}$ transferring the ticket to a coin. (If the ticket has already been spent, the receiver obtains a punish transaction $\mathsf{tx}_{\mathsf{pun}}$ that is used to revoke the deposit.) If the receiver does not tell the sender in a timely manner whether or not the protocol resulted in a nullpayment or a macropayment, the sender can refresh his ticket to obtain a refresh transaction $\mathsf{tx}_{\mathsf{ref}}$. Each party is responsible for publishing any output transaction ($\mathsf{tx}_{\mathsf{pt}}$, $\mathsf{tx}_{\mathsf{pun}}$, or $\mathsf{tx}_{\mathsf{ref}}$) by invoking $\mathbf{L}.\mathsf{Push}$ on it.

The receiver maintains two data structures across multiple probabilistic payments: (1) a *deposit blacklist* $\mathcal{D}$, which records the deposits that have been revoked within the current epoch (the ledger ensures that deposits revoked before the current epoch cannot be used to back payments); and (2) a *payment counter* $\mathcal{P}$, which keeps track of how many probabilistic payments and macropayments a customer has made within the current time window, to ensure that a customer does not 'overspend', violating bounds imposed by the economic analysis. For more details, see Section 6.3.

$\mathsf{DAM.PourCoinToCoin}^{\mathbf{L}}$	$\mathsf{DAM.PourCoinToDeposit}^{\mathbf{L}}$	$\mathsf{DAM.PourCoinToTicket}^{\mathbf{L}}$
public parameters pp old coins $[\mathbf{c}_i]_1^m$ old address secret keys $[\mathsf{ask}_i]_1^m$ new note values $[v_j]_1^n$ new address public keys $[\mathsf{apk}_j]_1^n$ public value v_{pub} transaction information string info		
—	receiver address sets $[\mathcal{R}_j]_1^n$	deposits $[\mathbf{d}_j]_1^n$
new coins $[\mathbf{c}_j]_1^n$ such that $\mathbf{c}_j.\mathsf{pub} = \mathtt{cn}$ $\mathbf{c}_j.\mathsf{sec} = \perp$	new deposits $[\mathbf{d}_j]_1^n$ such that $\mathbf{d}_j.\mathsf{pub} = \mathtt{dp}$ $\mathbf{d}_j.\mathsf{sec} =$ a commitment to $\mathcal{R}_j$	new tickets $[\mathbf{t}_j]_1^n$ such that $\mathbf{t}_j.\mathsf{pub} = \mathtt{tk}$ $\mathbf{t}_j.\mathsf{sec} = \mathbf{d}_j$
pour-coin transaction $\mathsf{tx}_{\mathsf{pc}}$		

$\mathsf{DAM.PourTicket}^{\mathbf{L}}$	
SENDER	RECEIVER
public parameters pp old ticket $\mathbf{t}$ old ticket address secret key $\mathsf{ask}_{\mathbf{t}}$ deposit $\mathbf{d}$ deposit address secret key $\mathsf{ask}_{\mathbf{d}}$ deposit receiver address set $\mathcal{R}_{\mathbf{d}}$ transaction information string info average-case and worst-case counters $\mathsf{actr}, \mathsf{wctr}$	public parameters pp current deposit blacklist $\mathcal{D}$ current payment counter $\mathcal{P}$ new coin address key pair $(\mathsf{apk}_{\mathbf{c}}, \mathsf{ask}_{\mathbf{c}})$ receiver time window tw_{r} public value v_{pub}
$\mathsf{status} \in \{\mathtt{null}, \mathtt{macro}, \mathtt{fail}\}$ if $\mathsf{status} = \mathtt{fail}$: also a new ticket $\mathbf{t}'$ & refresh transaction $\mathsf{tx}_{\mathsf{ref}}$	$\mathsf{status} \in \{\mathtt{null}, \mathtt{macro}, \mathtt{fail}, \mathtt{double}\}$ updated payment counter $\mathcal{P}'$ if $\mathsf{status} = \mathtt{double}$: also an updated blacklist $\mathcal{D}'$ & punish transaction $\mathsf{tx}_{\mathsf{pun}}$ if $\mathsf{status} = \mathtt{macro}$: also a new coin $\mathbf{c}$ & pour-ticket transaction $\mathsf{tx}_{\mathsf{pt}}$

Withdrawing deposits: A deposit can be *withdrawn*, pouring the deposit's value into a new coin, by using the algorithm DAM.WithdrawDeposit. Its output consists of the new coin $\mathbf{c}$ and a withdraw transaction $\mathsf{tx}_{\mathsf{wd}}$, which is to be published by invoking $\mathbf{L}.\mathsf{Push}(\mathsf{tx}_{\mathsf{wd}})$. Withdraw transactions have an activation delay of Δ_{w} so to prevent users from withdrawing a deposit before they can be punished for double spending tickets backed by it (see Section 6.3).

$\mathsf{DAM.WithdrawDeposit}^{\mathbf{L}}$
public parameters pp old deposit $\mathbf{d}$ old deposit address secret key $\mathsf{ask}_{\mathbf{d}}$ new coin address public key apk public value v_{pub} transaction information string info
new coin $\mathbf{c}$ withdraw transaction $\mathsf{tx}_{\mathsf{wd}}$

Refreshing tickets: A ticket can be *refreshed*, pouring the ticket's value into a new ticket, by using the algorithm $\mathsf{DAM.RefreshTicket}$. Its output consists of the new ticket $\mathbf{t}'$ and a refresh transaction $\mathsf{tx}_{\mathsf{ref}}$, which is to be published by invoking $\mathbf{L}.\mathsf{Push}(\mathsf{tx}_{\mathsf{ref}})$. Refresh transactions have an activation delay of Δ_{r} so to prevent users from spending tickets back to themselves (see Section 6.3). Refreshing a ticket enables a customer to safely overcome a merchant abort, i.e., when the merchant does not inform the customer of whether his probabilistic payment resulted in a nullpayment or a macropayment. Indeed, if the customer were to continue using the same ticket in the future, he may inadvertently double spend, and thereby lose his deposit. Note that refreshing tickets maintains ticket privacy, and thus malicious merchants cannot compromise the anonymity of an honest customer.

$\mathsf{DAM.RefreshTicket}^{\mathbf{L}}$
public parameters pp old ticket $\mathbf{t}$ old ticket address secret key $\mathsf{ask}_{\mathbf{t}}$ new ticket address public key apk public value v_{pub} transaction information string info
new ticket $\mathbf{t}'$ refresh transaction $\mathsf{tx}_{\mathsf{ref}}$

Punishing double spenders: $\mathsf{DAM.Punish}(\mathsf{tx}_{\mathsf{pt}}, \mathsf{tx}'_{\mathsf{pt}}, \mathcal{D}) \to (\mathsf{tx}_{\mathsf{pun}}, \mathcal{D}')$

On input two pour-ticket transactions $\mathsf{tx}_{\mathsf{pt}}$, $\mathsf{tx}'_{\mathsf{pt}}$ using the same ticket and the current deposit blacklist $\mathcal{D}$, $\mathsf{DAM.Punish}$ outputs a punish transaction revoking the deposit backing that ticket and an updated deposit blacklist $\mathcal{D}'$.

Verifying transactions: $\mathsf{DAM.VerifyTransaction}^{\mathbf{L}}(\mathsf{pp}, \mathsf{tx}) \to b$

On input public parameters pp and a transaction tx, and given oracle access to the ledger $\mathbf{L}$, $\mathsf{DAM.VerifyTransaction}$ output a bit b denoting whether the transaction tx is valid relative to the ledger $\mathbf{L}$.

6.3 Guidelines for usage

Since a DAM scheme has a rich interface, here we summarize how its various components are intended to interact. Before the system is used, a trusted third party runs $\mathsf{DAM.Setup}$ to generate the public parameters for the system, and publishes these for everyone to see. Afterwards, anyone can use $\mathsf{DAM.CreateAddr}$ to create his own address key pair, so as to mint notes and make payments; merchants must bind their payment rate, probability, and value to the address by passing these parameters as specification to $\mathsf{DAM.CreateAddr}$. One can obtain information about received payments by running $\mathsf{DAM.Receive}$. Throughout, anyone can use $\mathsf{DAM.VerifyTransaction}$ to verify a transaction; for example, this algorithm can be used by the nodes that run a distributed protocol to maintain the ledger (if the ledger is so maintained).

A safe use of probabilistic payments entails additional guidelines that customers and merchants must keep in mind.

- *Guidelines for customers.* A customer should remember that, while an honest merchant should respond right away, the merchant's response is not instantaneous; in the meantime, if the customer wishes to safely make additional probabilistic payments (even in parallel), he should *rotate* among different tickets. Yet, if a merchant response takes longer than expected, then the customer should interpret it as an 'abort' and *refresh* his ticket. (Or else a malicious

merchant could later frame the customer for double spending, or learn when the ticket is next spent.) Finally, a customer can *withdraw* his deposit if he no longer wishes to use that money to back tickets.

To bound the required value of a deposit, our economic analysis requires bounding the cumulative payment rate across the set of merchants that can be paid by payments backed by that deposit. Estimating and bounding the total number of merchants in a system is difficult to do at system setup time, and cautious over-estimation can lead to a unnecessary blow-up in the size of the deposit. To mitigate this, we associate with each deposit a set of 'valid' merchants (see Section 3.3). At deposit creation time, a user must specify such a *merchant set*, and set the value of the deposit accordingly. This allows a deposit to be only as large as the amount required to support financial activity with the merchants in the deposit's merchant set.

- *Guidelines for merchants.* The ledger partitions time into epochs and, in the current epoch, records 'history' up to and excluding the current epoch. While the ledger is updated only once per epoch, individual transactions propagate through the network at a much faster rate. For example, in Bitcoin an epoch is around 10 minutes but the network broadcast time is much smaller, on the order of seconds. Merchants can, and should, leverage information that they 'hear' from the network, in order to further limit malicious customers even if that information is not yet in the ledger.

  A merchant is responsible for broadcasting any pour-ticket transactions that he obtains as the result of engaging in probabilistic payments (as the receiver), in addition to publishing these to the ledger for the next epoch. Moreover, a merchant is responsible for maintaining two data structures across probabilistic payments, as we now explain.

  The merchant maintains a *deposit blacklist* $\mathcal{D}$, which records the deposits that have been revoked *within the current epoch*. (Deposits revoked in previous epochs are on the ledger, and cannot back probabilistic payments in this epoch.) The merchant resets $\mathcal{D}$ at the beginning of every epoch, and after that does the following.

  – Whenever the merchant engages in a probabilistic payment as the receiver, via the protocol $\mathsf{DAM.PourTicket}$, he provides the current $\mathcal{D}$ as one of the receiver inputs. In the event that the sender used a deposit that is blacklisted, the merchant also gets as output the updated $\mathcal{D}'$ and a punish transaction $\mathsf{tx}_{\mathsf{pun}}$, which he publishes to the ledger.

  – Whenever the merchant learns of a (valid) pour-ticket transaction $\mathsf{tx}'_{\mathsf{pt}}$ that shares the same serial number as another pour-ticket transaction $\mathsf{tx}_{\mathsf{pt}}$ seen in the current epoch, he runs $\mathsf{DAM.Punish}$ on input $\mathsf{tx}_{\mathsf{pt}}, \mathsf{tx}'_{\mathsf{pt}}$ and the current $\mathcal{D}$, and obtains as output the updated $\mathcal{D}'$ as well as a punish transaction $\mathsf{tx}_{\mathsf{pun}}$, which he publishes to the ledger.

  The merchant also maintains a *payment counter* $\mathcal{P}$, which records payment value rates in a merchant-defined time window, as we now explain. Our economic analysis (Section 3) says that the required deposit value is determined by either A or W, which denote the *per-deposit* maximum value of probabilistic payments or of macropayments (respectively) within a certain time period. Our scheme (and thus construction) support enforcement of both bounds simultaneously, but for simplicity in this discussion we focus on worst-case utility.

  A merchant M is responsible "for his own share": at address creation time, he chooses an worst-case bound w_{M}, and at protocol execution time he chooses a time window tw_{M} that partitions time into intervals of that length. His goal is to ensure that the per-deposit maximum value of probabilistic payments at every interval of length tw_{M} is at most w_{M}. Since which deposit backs a ticket is not revealed during a pour ticket, a DAM scheme provides functionality for the merchant to do so: whenever the merchant engages in a probabilistic payment as the receiver, via the protocol $\mathsf{DAM.PourTicket}$, he provides the current $\mathcal{P}$, and time window tw_{M} as receiver inputs, and obtains as output the updated $\mathcal{P}'$.

  Intuitively, if the sender's deposit is not at least w_{M}, the merchant M receives the output `fail`. Otherwise, the merchant can query the payment counter $\mathcal{P}$ to learn if the sender has breached the bound w_{M} in probabilistic payments with him. If so, the merchant should reject the sender's payment.

Side-channel information leakage. In sum, after each execution of the $\mathsf{DAM.PourTicket}$ protocol, the honest merchant learns at most the following information: whether the deposit is sufficient to back payments to the current merchant, and whether the rate limit for that merchant has been exceeded or not. However, a malicious merchant might attempt to glean extra information by repeating time windows; since rate limits are enforced within time windows, reusing time windows can reveal information about whether or not a customer has already interacted with the merchant in the past. To prevent this, the customer should keep track of past time windows, and refuse payment if these repeat.

(How many time windows the customer must store depends on the construction; our construction allows the customer to store only the time windows seen within the current ledger epoch and forget those of past epochs; see Section 7.2.3.)

Activation delays prevent front running. A DAM scheme enables users to withdraw deposits and refresh tickets. These operations could in principle create a security vulnerability: a malicious customer could attempt to front run the merchant by withdrawing a deposit before the merchant revokes it, or refresh a ticket before the merchant claims it. To prevent such scenarios, withdraw and refresh transactions have activation delays of Δ_{w} and Δ_{r}, i.e., they become active Δ_{w} and Δ_{r} epochs after they are added to the ledger. This gives time to a merchant to revoke the deposit or claim the ticket, if appropriate. If the merchant does so during the waiting period, then the withdraw or refresh transaction does not become active. (See Section 5.3 for more details on activation delays.)

6.4 Security

We informally describe the security goals of a DAM scheme (Section 6.4.1), and then formally state the security definition of a DAM scheme (Section 6.4.2) by specifying an ideal functionality $\mathcal{F}_{\mathrm{DAM}}$ that captures these goals.

6.4.1 Informal security goals

A DAM scheme provides functionality for deterministic payments and probabilistic payments, and we outline the security goals as they pertain to each of these.

Security of deterministic payments. Deterministic payments in a DAM scheme inherit the security of payments in a DAP scheme, including the following properties.

- *Transaction anonymity.* Transactions do not reveal a payment's origin, destination, or amount. Note, however, that transactions do *not* hide type information such as whether one is minting a coin, ticket, or deposit, or whether one is pouring from a coin, deposit, or ticket; overall, we do not seek to hide type transitions.
- *Satisfaction of financial invariants.* Users cannot own or spend more money than they have obtained by minting new notes or receiving notes from other users. In particular, users cannot spend the same note multiple times.
- *Transaction non-malleability.* Malicious parties cannot modify a transaction 'in flight' before it reaches the ledger.

Security of probabilistic payments. Probabilistic payments between senders and receivers enrich payment semantics and introduce interaction, and thus entail new security goals that are not captured by the above.

- *Sender privacy.* Probabilistic payments of honest senders are unlinkable, not only given information on the ledger, but also from the perspective of receivers; this holds regardless of whether the payment turns into a macropayment or nullpayment. More precisely, both macropayments and nullpayments reveal no information about which tickets are used, and the receiver learns only that the deposit backing the payment is large enough to support the sender's spending activity. (Naturally, if a sender double spends, his anonymity is not guaranteed.)
- *Sender utility.* If an honest sender engages with a malicious receiver in a probabilistic payment, the malicious receiver cannot force the sender (even by aborting) to lose his deposit or anonymity, or to lose his ticket with any probability other than the specified one for the payment.
- *Receiver privacy.* Engaging in a probabilistic payment as a receiver does not reveal to the ledger any receiver-specific information, such as the receiver address, payment probability, macropayment value, or payment value rate.
- *Receiver utility (against rational senders).* We cannot guarantee that a receiver, who engages with a malicious sender in a probabilistic payment, will not lose money, because a malicious sender may double spend regardless of the negative utility incurred by losing his deposit. At best, we can require that malicious senders *will* get their deposits revoked when engaging with honest receivers. We thus require that: (1) each ticket is bound to a single deposit; (2) revoked deposits cannot be withdrawn; (3) spent tickets cannot be refreshed; (4) senders cannot bypass payment rate bounds; (5) deposits cannot be withdrawn before an activation delay of Δ_{w} epochs; (6) tickets cannot be refreshed before an activation delay of Δ_{r} epochs; (7) tickets with identical serial numbers must have the same deposit; (8) generating two macropayments backed by the same ticket reveals the deposit.

6.4.2 Security definition

We define security of a DAM scheme via the real/ideal paradigm [Gol04]: we specify an ideal functionality $\mathcal{F}_{\mathrm{DAM}}$ and say that a DAM scheme is *secure* if it realizes $\mathcal{F}_{\mathrm{DAM}}$ against static corruptions. We focus on *standalone* security, and

leave achieving security under composition to future work (which may take [KMS+16] as a starting point).

Definition 6.1 (security of a DAM scheme)**.** *A DAM scheme* DAM *is* **secure** *if for every honest execution strategy* Σ *and efficient real-world adversary* $\mathcal{A}$ *that corrupts a subset of parties, there exists an efficient ideal-world simulator* $\mathsf{Sim}_{\mathcal{A}}$ *that corrupts the same subset of parties such that the view of* $\mathcal{A}$ *when running in a real-world execution of* DAM *is computationally indistinguishable from the output of* $\mathsf{Sim}_{\mathcal{A}}$ *when running in an ideal-world execution with the ideal functionality* $\mathcal{F}_{\mathrm{DAM}}$.

The ideal functionality. The ideal functionality $\mathcal{F}_{\mathrm{DAM}}$ is specified in Figures 1 and 2. The ideal functionality maintains in clk (initialized to 0) the current epoch, and maintains the following tables (initialized as empty): (1) Addr, which maps an address to the party that owns it; (2) Notes, which maps a note identifier to that note's information, consisting of the note's type, value, address, and activation time; (3) State, which maps a note identifier to that note's state, either `unspent` or `spent`; (4) Buf, which temporarily holds coins and tickets that result from deposit withdrawal or ticket refreshes (as these have positive activation delays); (5) Spent, which counts the number of times a ticket has been spent to a particular session identifier sid; (6) Rates, which counts the number of probabilistic payments backed by a deposit $\mathbb{d}$ that have been made in a given time window.

Parties may obtain addresses from the ideal functionality. Parties can mint notes and receive poured notes to these addresses. Each note can be in one of two states: `unspent` or `spent`. When transactions occur, the ideal functionality sends to all parties the identifier of the notes that were consumed, the public output, and the number of input and output notes. If concurrent calls are made to the ideal functionality, it serializes them; the only exception is calls to PourTicket, which are serialized only up till Step 27 (in order to allow interleaving of interactions).

Operation of the honest parties. In both the real and ideal worlds, each *uncorrupted* party is provided with an adaptive execution strategy $\Sigma^{\mathbf{L}}$ that informs its operation. In both worlds, the execution strategy can invoke the various mint and pour algorithms, initiate pour-ticket protocol executions, withdraw deposits, refresh tickets, and retrieve unspent coins. In return, the execution strategy receives a notification when a request has been satisfied, and whenever a transaction has been conducted by another party. In both worlds we define an honest party that runs the execution strategy in such a way that the structure of inputs to the executions strategy is identical between the two worlds. A full description of the operation of these parties is provided in Appendix C.

$\mathsf{RegisterAddress}[\mathcal{P}]\begin{pmatrix}\text{address identifier } \mathsf{id},\\ \text{probabilistic payment spec } \mathsf{spec}\end{pmatrix}$	$\mathsf{IncrementEpoch}[\mathcal{P}]$	$\mathsf{GetTime}[\mathcal{P}]$
1. Set $\mathsf{a} := (\mathsf{id}\|\mathsf{spec})$. 2. If $\mathsf{Addr}[\mathsf{a}] \neq \perp$: **Send to $\mathcal{P}$:** $\mathtt{registered}$. 3. If $\mathsf{Addr}[\mathsf{a}] = \perp$: (a) set $\mathsf{Addr}[\mathsf{a}] := \mathcal{P}$. (b) **Send to $\mathcal{P}$:** address a.	1. Check that all parties have invoked $\mathsf{IncrementEpoch}$. 2. Set $\mathsf{clk} := \mathsf{clk} + 1$. 3. Reset Rates and $\mathsf{DBTable}$ to be empty. 4. $\mathsf{ProcessBuffer}()$.	1. **Send to $\mathcal{P}$:** clk.

$\mathsf{MintCoin}[\mathcal{P}]\begin{pmatrix}\text{new value } v,\\ \text{new address } \mathsf{a}\end{pmatrix}$	$\mathsf{MintDeposit}[\mathcal{P}]\begin{pmatrix}\text{new value } v,\\ \text{new address } \mathsf{a},\\ \text{receiver address set } \mathcal{R}\end{pmatrix}$	$\mathsf{MintTicket}[\mathcal{P}]\begin{pmatrix}\text{new value } v,\\ \text{new address } \mathsf{a},\\ \text{deposit information } \mathsf{info}^{\mathfrak{d}}\end{pmatrix}$
1. Sample unique coin identifier $\mathfrak{c}$. 2. Set $\mathsf{Notes}[\mathfrak{c}] := (\mathtt{cn}, v, \mathsf{a}, \mathsf{clk})$. 3. Set $\mathsf{State}[\mathfrak{c}] := \mathtt{unspent}$. 4. Set $\mathsf{Token}[\mathfrak{c}] := \mathsf{tok}_{\mathtt{cn}}$. 5. **Send to all parties:** $(\mathtt{mintcn}, \mathfrak{c}, v)$.	1. Sample unique deposit identifier $\mathfrak{d}$. 2. Set $\mathsf{Notes}[\mathfrak{d}] := (\mathtt{dp}, v, \mathsf{a}, \mathsf{clk}, \mathcal{R})$. 3. Set $\mathsf{State}[\mathfrak{d}] := \mathtt{unspent}$. 4. Set $\mathsf{State}[\mathfrak{d}] := \mathsf{tok}_{\mathtt{dp}}$. 5. **Send to all parties:** $(\mathtt{mintdp}, \mathfrak{d}, v)$.	1. Check that $\mathsf{info}^{\mathfrak{d}}.\mathsf{a}_{\mathfrak{d}} = \mathsf{a}$. 2. Sample unique ticket identifier $\mathfrak{t}$. 3. Set $\mathsf{Notes}[\mathfrak{t}] := (\mathtt{tk}, v, \mathsf{a}, \mathsf{clk}, \mathsf{info}^{\mathfrak{d}})$. 4. Set $\mathsf{State}[\mathfrak{t}] := \mathtt{unspent}$. 5. Set $\mathsf{State}[\mathfrak{t}] := \mathsf{tok}_{\mathtt{tk}}$. 6. **Send to all parties:** $(\mathtt{minttk}, \mathfrak{t}, v)$.

$\mathsf{PourCoinToCoin}[\mathcal{P}]\begin{pmatrix}\text{old coins } [\mathfrak{c}_i]_1^m,\\ \text{new addr. } [\mathsf{a}_j]_1^n,\\ \text{new values } [v_j]_1^n,\\ \text{public value } v_{\mathsf{pub}}\end{pmatrix}$	$\mathsf{PourCoinToDeposit}[\mathcal{P}]\begin{pmatrix}\text{old coins } [\mathfrak{c}_i]_1^m,\\ \text{new addr. } [\mathsf{a}_j]_1^n,\\ \text{new values } [v_j]_1^n,\\ \text{public value } v_{\mathsf{pub}},\\ \text{receiver addr. sets } [\mathcal{R}_j]_1^n\end{pmatrix}$	$\mathsf{PourCoinToTicket}[\mathcal{P}]\begin{pmatrix}\text{old coins } [\mathfrak{c}_i]_1^m,\\ \text{new addr. } [\mathsf{a}_j]_1^n,\\ \text{new values } [v_j]_1^n,\\ \text{public value } v_{\mathsf{pub}},\\ \text{deposits } [\mathsf{info}^{\mathfrak{d}}{}_j]_1^n\end{pmatrix}$
	1. Check that $\sum_{i=1}^m v_i = \sum_{j=1}^n v_j + v_{\mathsf{pub}}$. 2. For each $i \in \{1, \ldots, m\}$: (a) check that $\mathsf{State}[\mathfrak{c}_i] = \mathtt{unspent}$. (b) retrieve $(\mathsf{tp}_i, v_i, \mathsf{a}_i, t_i) := \mathsf{Notes}[\mathfrak{c}_i]$. (c) check that $\mathsf{tp}_i = \mathtt{cn}$. (d) check that $\mathsf{Addr}[\mathsf{a}_i] = \mathcal{P}$. (e) set $\mathsf{State}[\mathfrak{c}_i] := \mathtt{spent}$.	
3. For each $j \in \{1, \ldots, n\}$: (a) sample unique coin identifier $\mathfrak{c}_j$. (b) set $\mathsf{Notes}[\mathfrak{c}_j] := (\mathtt{cn}, v_j, \mathsf{a}_j, \mathsf{clk})$. (c) set $\mathsf{State}[\mathfrak{c}_j] := \mathtt{unspent}$. (d) if $\exists\ \mathcal{P}_j$ s.t. $\mathsf{Addr}[\mathsf{a}_j] = \mathcal{P}_j$: **Send to $\mathcal{P}_j$:** $(\mathtt{prvpc}, \mathtt{cn}, \mathfrak{c}_j, v_j, \mathsf{a}_j)$. (e) otherwise: **Send to $\mathcal{A}$:** $(\mathtt{prvpc}, \mathtt{cn}, \mathfrak{c}_j, v_j, \mathsf{a}_j)$. 4. **Send to all parties:** $(\mathtt{pourcn}, m, [\mathfrak{c}_j]_1^n, v_{\mathsf{pub}})$.	3. For each $j \in \{1, \ldots, n\}$: (a) sample unique deposit identifier $\mathfrak{d}_j$. (b) set $\mathsf{Notes}[\mathfrak{d}_j] := (\mathtt{dp}, v_j, \mathsf{a}_j, \mathsf{clk}, \mathcal{R}_j)$. (c) set $\mathsf{State}[\mathfrak{d}_j] := \mathtt{unspent}$. (d) if $\exists\ \mathcal{P}_j$ s.t. $\mathsf{Addr}[\mathsf{a}_j] = \mathcal{P}_j$: **Send to $\mathcal{P}_j$:** $(\mathtt{prvpc}, \mathtt{dp}, \mathfrak{d}_j, v_j, \mathsf{a}_j, \mathcal{R}_j)$. (e) otherwise: **Send to $\mathcal{A}$:** $(\mathtt{prvpc}, \mathtt{dp}, \mathfrak{d}_j, v_j, \mathsf{a}_j, \mathcal{R}_j)$. 4. **Send to all parties:** $(\mathtt{pourcn}, m, [\mathfrak{d}_j]_1^n, v_{\mathsf{pub}})$.	3. For each $j \in \{1, \ldots, n\}$: (a) check that for all $i = 1$ to m: $\mathsf{a}_j = \mathsf{a}_i = \mathsf{info}_j^{\mathfrak{d}}.\mathsf{a}_{\mathfrak{d}}$. (b) sample unique ticket identifier $\mathfrak{t}_j$. (c) set $\mathsf{Notes}[\mathfrak{t}_j] := (\mathtt{tk}, v_j, \mathsf{a}_j, \mathsf{clk}, \mathsf{info}_j^{\mathfrak{d}})$. (d) set $\mathsf{State}[\mathfrak{t}_j] := \mathtt{unspent}$. (e) if $\exists\ \mathcal{P}_j$ s.t. $\mathsf{Addr}[\mathsf{a}_j] = \mathcal{P}_j$: **Send to $\mathcal{P}_j$:** $(\mathtt{prvpc}, \mathtt{tk}, \mathfrak{t}_j, v_j, \mathsf{a}_j, \mathsf{info}_j^{\mathfrak{d}})$. (f) otherwise: **Send to $\mathcal{A}$:** $(\mathtt{prvpc}, \mathtt{tk}, \mathfrak{t}_j, v_j, \mathsf{a}_j, \mathsf{info}_j^{\mathfrak{d}})$. 4. **Send to all parties:** $(\mathtt{pourcn}, m, [\mathfrak{t}_j]_1^n, v_{\mathsf{pub}})$.

$\mathsf{WithdrawDeposit}[\mathcal{P}]\begin{pmatrix}\text{deposit } \mathfrak{d},\\ \text{new address } \mathsf{a}_{\mathsf{new}},\\ \text{public value } v_{\mathsf{pub}}\end{pmatrix}$	$\mathsf{RefreshTicket}[\mathcal{P}]\begin{pmatrix}\text{ticket } \mathfrak{t},\\ \text{new address } \mathsf{a}_{\mathsf{new}},\\ \text{public value } v_{\mathsf{pub}}\end{pmatrix}$	$\mathsf{ProcessBuffer}()$
1. Retrieve $(\mathtt{dp}, v, \mathsf{a}, t, \mathcal{R}) := \mathsf{Notes}[\mathfrak{d}]$. 2. Check that $\mathsf{State}[\mathfrak{d}] = \mathtt{unspent}$. 3. Check that $\mathsf{Addr}[\mathsf{a}] = \mathcal{P}$. 4. Set $\mathsf{Buf}[\mathfrak{d}, *] := \perp$. 5. Sample unique coin identifier $\mathfrak{c}$. 6. Set $v' := v - v_{\mathsf{pub}}$. 7. Set $t' := \mathsf{clk} + \Delta_{\mathsf{w}}$. 8. Set $\mathsf{Buf}[\mathbf{d}, t'] := (\mathfrak{c}, (\mathtt{cn}, v', \mathsf{a}_{\mathsf{new}}, t'))$. 9. If $\exists\ \mathcal{P}'$ s.t. $\mathsf{Addr}[\mathsf{a}_{\mathsf{new}}] = \mathcal{P}'$: **Send to $\mathcal{P}'$:** $(\mathtt{withdraw}, \mathtt{cn}, \mathfrak{c}, v, \mathsf{a}_{\mathsf{new}})$. 10. Otherwise: **Send to $\mathcal{A}$:** $(\mathtt{withdraw}, \mathtt{cn}, \mathfrak{c}, v, \mathsf{a}_{\mathsf{new}})$. 11. **Send to all parties:** $(\mathtt{withdraw}, \mathfrak{c}, v_{\mathsf{pub}})$.	1. Retrieve $(\mathtt{tk}, v, \mathsf{a}, t, \mathsf{info}^{\mathfrak{d}}) := \mathsf{Notes}[\mathfrak{t}]$. 2. Check that $\mathsf{a}_{\mathsf{new}} = \mathsf{a}$. 3. Check that $\mathsf{State}[\mathfrak{t}] = \mathtt{unspent}$. 4. Check that $\mathsf{Addr}[\mathsf{a}] = \mathcal{P}$. 5. Set $\mathsf{Buf}[\mathfrak{t}, *] := \perp$. 6. Sample unique ticket identifier $\mathfrak{t}'$. 7. Set $v' := v - v_{\mathsf{pub}}$. 8. Set $t' := \mathsf{clk} + \Delta_{\mathsf{r}}$. 9. Set $\mathsf{Buf}[\mathfrak{t}, t'] := (\mathfrak{t}', (\mathtt{tk}, v', \mathsf{a}_{\mathsf{new}}, t', \mathsf{info}^{\mathfrak{d}}))$. 10. If $\exists\ \mathcal{P}'$ s.t. $\mathsf{Addr}[\mathsf{a}_{\mathsf{new}}] = \mathcal{P}'$: **Send to $\mathcal{P}'$:** $(\mathtt{refresh}, \mathtt{tk}, \mathfrak{t}', v', \mathsf{a}_{\mathsf{new}}, \mathsf{info}^{\mathfrak{d}})$. 11. Otherwise: **Send to $\mathcal{A}$:** $(\mathtt{refresh}, \mathtt{tk}, \mathfrak{t}', v', \mathsf{a}_{\mathsf{new}}, \mathsf{info}^{\mathfrak{d}})$. 12. **Send to all parties:** $(\mathtt{refresh}, \mathfrak{t}', v_{\mathsf{pub}})$.	1. If there exists $\mathfrak{d}$ such that $\mathsf{Buf}[\mathfrak{d}, \mathsf{clk}] \neq \perp$: (a) Retrieve $(\mathfrak{c}, (\mathtt{cn}, v, \mathsf{a}, t)) := \mathsf{Buf}[\mathfrak{d}, \mathsf{clk}]$. (b) Check that $\mathsf{State}[\mathfrak{d}] = \mathtt{unspent}$. (c) Set $\mathsf{State}[\mathfrak{d}] := \mathtt{spent}$. (d) Set $\mathsf{Notes}[\mathfrak{c}] := (\mathtt{cn}, v, \mathsf{a}, t)$. (e) Set $\mathsf{State}[\mathfrak{c}] := \mathtt{unspent}$. 2. If there exists $\mathfrak{t}$ such that $\mathsf{Buf}[\mathfrak{t}, \mathsf{clk}] \neq \perp$: (a) Retrieve $(\mathfrak{t}', (\mathtt{tk}, v, \mathsf{a}, t, \mathsf{info}^{\mathfrak{d}})) := \mathsf{Buf}[\mathfrak{t}, \mathsf{clk}]$. (b) Check that $\mathsf{State}[\mathfrak{t}] = \mathtt{unspent}$. (c) Set $\mathsf{State}[\mathfrak{t}] := \mathtt{spent}$. (d) Set $\mathsf{Notes}[\mathfrak{t}'] := (\mathtt{tk}, v, \mathsf{a}, t, \mathsf{info}^{\mathfrak{d}})$. (e) Set $\mathsf{State}[\mathfrak{t}'] := \mathtt{unspent}$.

Figure 1: Ideal functionality $\mathcal{F}_{\mathrm{DAM}}$ of a DAM scheme (part 1 of 2).

$\mathsf{PourTicket}[\mathcal{P}_\mathsf{s}]$ (receiver party $\mathcal{P}_\mathsf{r}$)	$\mathsf{RedeemTicket}[\mathcal{P}_\mathsf{r}]$ (ticket token $\mathsf{tok}_\mathtt{tk}$, session identifier sid, public value v_pub)
1. **Send to $\mathcal{P}_\mathsf{r}$:** $\mathtt{init}$. 2. **Receive from $\mathcal{P}_\mathsf{r}$:** $\mathsf{sid}, \mathcal{D}, v_\mathsf{pub}$. 3. Parse sid as $(\mathsf{a}_\mathsf{r}, \mathsf{tw}_\mathsf{r})$. 4. Parse a_r as $(*\|(\mathfrak{a}_\mathsf{r}, \mathfrak{w}_\mathsf{r}, p_\mathsf{r}, V_\mathsf{r}))$. 5. **Send to $\mathcal{P}_\mathsf{s}$:** $(\mathsf{sid}, v_\mathsf{pub})$. 6. **Receive from $\mathcal{P}_\mathsf{s}$:** $\mathtt{ok}?$. 7. If $\mathtt{ok}? = \perp$: halt and **Send to $\mathcal{P}_\mathsf{r}$:** $\mathtt{abort}$. 8. If $\mathtt{ok}? = \mathbb{t}$: continue. 9. Retrieve $(\mathtt{tk}, v, \mathsf{a}, t, \mathsf{info}^\mathbb{d}) := \mathsf{Notes}[\mathbb{t}]$. 10. Parse $\mathsf{info}^\mathbb{d}$ as $(\mathbb{d}', v'_\mathbb{d}, \mathsf{a}'_\mathbb{d})$. 11. Check that $\mathsf{Addr}[\mathsf{a}] = \mathcal{P}_\mathsf{s}$. CHECK THAT DEPOSIT IS VALID AND UNSPENT. 12. Check that $\mathsf{State}[\mathbb{d}] = \mathtt{unspent}$. 13. Retrieve $(\mathsf{dp}, v_\mathbb{d}, \mathsf{a}_\mathbb{d}, t_\mathbb{d}, \mathcal{R}_\mathbb{d}) := \mathsf{Notes}[\mathbb{d}]$ 14. Check that $(\mathbb{d}', v'_\mathbb{d}, \mathsf{a}'_\mathbb{d}) = (\mathbb{d}, v_\mathbb{d}, \mathsf{a}_\mathbb{d})$. 15. Check that $\mathsf{Addr}[\mathsf{a}_\mathbb{d}] = \mathcal{P}_\mathsf{s}$. CHECK IF DEPOSIT WAS BLACKLISTED IN CURRENT EPOCH. 16. Retrieve $\mathsf{tok}_\mathsf{dp} := \mathsf{Token}[\mathbb{d}]$. 17. Check that $\mathsf{DBTable}[\mathbf{d}] = \mathtt{False}$ and $\mathsf{tok}_\mathsf{dp} \notin \mathcal{D}$. CHECK THAT DEPOSIT IS SUFFICIENTLY LARGE. 18. Set $A := \sum_{\mathsf{a}_i \in \mathcal{R}_\mathbb{d}} \mathsf{a}_i.\mathfrak{a}$. 19. Set $W := \sum_{\mathsf{a}_i \in \mathcal{R}_\mathbb{d}} \mathsf{a}_i.\mathfrak{w}$. 20. Set $V_\mathsf{max} := \max_{\mathsf{a}_i \in \mathcal{R}_\mathbb{d}} (\mathsf{a}_i.V)$. 21. If $\mathsf{a}_\mathsf{r} \notin \mathcal{R}_\mathbb{d}$: (a) if $W + \mathfrak{w}_\mathsf{r} \leq v_\mathbb{d}$ and $A + \mathfrak{a}_\mathsf{r} + \max(V_\mathsf{max}, V_\mathsf{r}) \leq v_\mathbb{d}$: set $\mathsf{Notes}[\mathbb{d}] := (\mathsf{dp}, v_\mathbb{d}, \mathsf{a}_\mathbb{d}, t_\mathbb{d}, \mathcal{R}_\mathbb{d} \cup \{\mathsf{a}_\mathsf{r}\})$. (b) else: halt. CHECK AVERAGE-CASE PAYMENT VALUE RATE BOUND. 22. Set $\mathsf{Rates}[\mathtt{A}, \mathbb{d}, \mathsf{sid}] := \mathsf{Rates}[\mathtt{A}, \mathbb{d}, \mathsf{sid}] + 1$. 23. If $\mathsf{Rates}[\mathtt{A}, \mathbb{d}, \mathsf{sid}] \leq \mathfrak{a}_\mathsf{r}/p_\mathsf{r}V_\mathsf{r}$: set $\mathsf{avg} := 0$. 24. If $\mathsf{Rates}[\mathtt{A}, \mathbb{d}, \mathsf{sid}] > \mathfrak{a}_\mathsf{r}/p_\mathsf{r}V_\mathsf{r}$: set $\mathsf{avg} := 1$. 25. Set $b := 1$ with probability p, and 0 otherwise. DECIDE PAYMENT OUTCOME. 26. If $b = 0$: set $\mathsf{result} := \mathtt{null}$. 27. If $b = 1$: CHECK WORST-CASE PAYMENT VALUE RATE BOUND. (a) set $\mathsf{Rates}[\mathtt{W}, \mathbb{d}, \mathsf{sid}] := \mathsf{Rates}[\mathtt{A}, \mathbb{d}, \mathsf{sid}] + 1$. (b) if $\mathsf{Rates}[\mathtt{W}, \mathbb{d}, \mathsf{sid}] \leq \mathfrak{w}_\mathsf{r}/p_\mathsf{r}V_\mathsf{r}$: set $\mathsf{worst} := 0$. (c) if $\mathsf{Rates}[\mathtt{W}, \mathbb{d}, \mathsf{sid}] > \mathfrak{w}_\mathsf{r}/p_\mathsf{r}V_\mathsf{r}$: set $\mathsf{worst} := 1$. CHECK DOUBLE SPENDING. (d) retrieve $\mathsf{tok}_\mathtt{tk} := \mathsf{Token}[\mathbb{t}]$. (e) if $\mathsf{Spent}[\mathbb{t}, *, *] = 0$: set $\mathsf{result} := (\mathtt{macro}, \mathsf{tok}_\mathtt{tk})$. (f) if $\mathsf{Spent}[\mathbb{t}, *, *] \geq 1$: i. set $\mathsf{result} := (\mathtt{macro}, \mathsf{tok}_\mathtt{tk}, \mathsf{tok}_\mathsf{dp})$. ii. set $\mathsf{DBTable}[\mathbf{d}] := \mathtt{True}$. (g) set $\mathsf{Spent}[\mathbb{t}, \mathsf{sid}, v_\mathsf{pub}] := \mathsf{Spent}[\mathbb{t}, \mathsf{sid}, v_\mathsf{pub}] + 1$. 28. **Send to $\mathcal{P}_\mathsf{r}$:** $(\mathtt{pourtk}, \mathsf{avg}, \mathsf{worst}, \mathsf{result})$. 29. **Receive from $\mathcal{P}_\mathsf{r}$:** $\mathtt{ok}?$. 30. If $\mathtt{ok}? = 0$: **Send to $\mathcal{P}_\mathsf{s}$:** $\mathtt{abort}$. 31. If $\mathtt{ok}? = 1$: **Send to $\mathcal{P}_\mathsf{s}$:** b.	1. Check that $\exists \mathbb{t}$ s.t. $\mathsf{tok}_\mathtt{tk} := \mathsf{Token}[\mathbb{t}]$. 2. If so, retrieve $\mathbb{t}$. 3. Check that $\mathsf{Spent}[\mathbb{t}, \mathsf{sid}, v_\mathsf{pub}] \geq 1$. 4. Parse sid as $(\mathsf{a}_\mathsf{r}, \mathsf{tw}_\mathsf{r})$. 5. Retrieve $(\mathtt{tk}, v, \mathsf{a}, t, \mathsf{info}^\mathbb{d}) := \mathsf{Notes}[\mathbb{t}]$. 6. If $\mathsf{State}[\mathbb{t}] = \mathtt{unspent}$: (a) set $\mathsf{State}[\mathbb{t}] := \mathtt{spent}$. (b) sample unique coin identifier $\mathbb{c}$. (c) set $\mathsf{Notes}[\mathbb{c}] := (\mathtt{cn}, v - v_\mathsf{pub}, \mathsf{a}_\mathsf{r}, \mathsf{clk})$. (d) If $\exists\ \mathcal{P}$ such that $\mathsf{Addr}[\mathsf{a}_\mathsf{r}] = \mathcal{P}$: **Send to $\mathcal{P}$:** $(\mathtt{ptk}, \mathtt{cn}, \mathbb{c}, v - v_\mathsf{pub}, \mathsf{a}_\mathsf{r})$. (e) Otherwise: **Send to $\mathcal{A}$:** $(\mathtt{ptk}, \mathtt{cn}, \mathbb{c}, v - v_\mathsf{pub}, \mathsf{a}_\mathsf{r})$. (f) **Send to $\mathcal{P}_\mathsf{r}$:** $(\mathtt{macropay})$. (g) **Send to all parties:** $(\mathtt{pourtk}, \mathbb{c}, v_\mathsf{pub})$. 7. If $\mathsf{State}[\mathbb{t}] = \mathtt{spent}$ and $\mathsf{Spent}[\mathbb{t}, *, *] = 1$: (a) **Send to $\mathcal{P}_\mathsf{r}$:** $\mathtt{toolate}$. 8. If $\mathsf{State}[\mathbb{t}] = \mathtt{spent}$ and $\mathsf{Spent}[\mathbb{t}, *, *] \geq 1$: (a) set $\mathsf{State}[\mathbb{d}] := \mathtt{spent}$. (b) retrieve $\mathsf{tok}_\mathsf{dp} := \mathsf{Token}[\mathbb{d}]$. (c) **Send to all parties:** $(\mathtt{doublespend}, \mathsf{tok}_\mathsf{dp})$.

Figure 2: Ideal functionality $\mathcal{F}_\mathrm{DAM}$ of a DAM scheme (part 2 of 2).

7 Construction of a decentralized anonymous micropayment scheme

We present our construction of a decentralized anonymous micropayment (DAM) scheme (see Section 6 for its definition); the main building blocks are DAP schemes (see Section 5) and FMT schemes (see Section 4). This section is organized as follows: in Section 7.1 we informally describe our construction; in Section 7.2 we describe additional building blocks; in Section 7.3 we give the pseudocode of our construction (see Figures 3, 4, 5); in Section 7.4 we state the theorem that asserts security, and its proof is in Appendix C.

7.1 Informal description

Recall that a DAM scheme is a tuple of algorithms:

$$\mathsf{DAM} = \left(\begin{array}{lllll} \mathsf{Setup} & \mathsf{MintCoin}^{\mathbf{L}} & \mathsf{PourCoinToCoin}^{\mathbf{L}} & & \\ \mathsf{CreateAddr} & \mathsf{MintDeposit}^{\mathbf{L}} & \mathsf{PourCoinToDeposit}^{\mathbf{L}} & \mathsf{WithdrawDeposit}^{\mathbf{L}} & \mathsf{Punish} \\ \mathsf{Receive}^{\mathbf{L}} & \mathsf{MintTicket}^{\mathbf{L}} & \mathsf{PourCoinToTicket}^{\mathbf{L}} & \mathsf{RefreshTicket}^{\mathbf{L}} & \mathsf{VerifyTransaction}^{\mathbf{L}} \\ & & \mathsf{PourTicket}^{\mathbf{L}} & & \end{array} \right).$$

We sketch the construction of these in Section 7.1.1 and Section 7.1.2, and then separately discuss security intuition in Section 7.1.3 and 'pour regulation' in Section 7.1.4.

7.1.1 Informal algorithm descriptions

Setup. The algorithm DAM.Setup samples public parameters for the various building blocks that we use, which includes DAP schemes and FMT schemes, as well as one-time signature schemes and NIZKs (see Section 7.2).

Creating addresses. The algorithm DAM.CreateAddr samples a new address key pair by running DAP.CreateAddr with the address information set to the probabilistic payment specification and outputting its result; in other words, DAM addresses are simply addresses of the underlying DAP scheme. Receivers must bind their intended payment rates, probability, and value to the address by passing it as input to DAM.CreateAddr (other users can set it to $\perp$ if they do not intend to receive probabilistic payments).

Receiving coins. The algorithm DAM.Receive, given an address key pair, retrieves all the unspent coins sent to this address by simply running DAP.Receive. Indeed, DAM pour-coin, pour-ticket, withdraw, and refresh transactions can be viewed as DAP pour transactions, and so DAP.Receive may retrieve from these any relevant coins.

Minting notes. Each of the minting algorithms DAM.MintCoin, DAM.MintDeposit, and DAM.MintTicket first sets the public information string pub to the type of the note being minted (respectively, `cn`, `dp`, or `tk`), and sets the secret information string sec accordingly: for coins, sec equals $\perp$; for deposits, sec is a commitment to the deposit's receiver address set $\mathcal{R}$; for tickets, sec equals the (already-minted) deposit that backs it. Then, the algorithm mints the note by running DAP.Mint.

Pouring coins. Each of the algorithms DAM.PourCoinToCoin, DAM.PourCoinToDeposit, DAM.PourCoinToTicket first sets the public and secret information strings similarly to above, and then runs DAP.Pour to generate the new notes. A DAM scheme also includes a protocol for pouring tickets into coins, which we discuss separately in Section 7.1.2 because it is the most complex part of the construction.

Withdrawing deposits. The algorithm DAM.WithdrawDeposit, given a deposit $\mathbf{d}$ (and its address secret key) and address public key apk, pours the deposit into a new coin $\mathbf{c}$ with address apk by running DAP.Pour. The output consists of the new coin $\mathbf{c}$, as well as a withdraw transaction $\mathsf{tx}_{\mathsf{wd}}$ that is just a DAP pour transaction having activation delay Δ_{w}. Since pour transactions reveal the serial numbers of input notes, it is easy to blacklist the withdrawn deposit (see Section 7.1.2).

Refreshing tickets. The algorithm DAM.RefreshTicket, given a ticket $\mathbf{t}$ (and its address secret key) and address public key apk, pours the ticket into a new one with address apk by using DAP.Pour; no information about the ticket (except its serial number) is revealed in a refresh transaction. The output pour transaction has activation delay Δ_{r}. (Recall that ticket refreshes enable honest users to avoid double spending when dealing with malicious merchants; see Section 6.3.)

Punishing double spenders. The algorithm DAM.Punish, given as input two pour-ticket transactions $\mathsf{tx}_{\mathsf{pt}}$ and $\mathsf{tx}'_{\mathsf{pt}}$ and a deposit blacklist, recovers from the pour-ticket transactions the serial number sn and identifier id of the deposit that backed both payments and adds it to the deposit blacklist. It then outputs a punish transaction $\mathsf{tx}_{\mathsf{pun}}$ consisting of $(\mathsf{tx}_{\mathsf{pt}}, \mathsf{tx}'_{\mathsf{pt}}, \mathsf{sn}, \mathsf{id})$. Once $\mathsf{tx}_{\mathsf{pun}}$ appears on the ledger, $\mathbf{d}$'s serial number becomes public, and $\mathbf{d}$ is considered revoked.

Verifying transactions. The algorithm DAM.VerifyTransaction, given a transaction tx, conducts different verifications on tx depending on its type.

1. *Mint transaction.* DAM.VerifyTransaction passes the transaction directly to DAP.VerifyTransaction.

2. *Pour-coin, withdraw, and refresh transactions.* DAM.VerifyTransaction first checks that the type specified in the transaction information string info of tx contains the correct type for the input notes, and then passes tx to DAP.VerifyTransaction.

3. *Pour-ticket transaction.* DAM.VerifyTransaction retrieves from tx a "pour-ticket" zero-knowledge proof (which we discuss later in Section 7.2.7) and verifies it, and also retrieves from tx a DAP pour transaction and uses DAP.VerifyTransaction to verify it.

4. *Punish transaction.* Punish transactions contain two pour-ticket transactions (allegedly both about the same double-spent ticket) and a deposit $\mathbf{d}$ (allegedly backing that ticket); the DAM.VerifyTransaction verifies that both pour-ticket transactions are valid and, moreover, that one can recover $\mathbf{d}$ from these two by combining two shares of $\mathbf{d}$ contained in each of these pour-ticket transactions (we discuss later in Section 7.2.5 how we do this secret sharing).

7.1.2 A 3-message protocol for probabilistic payments

We outline the construction of DAM.PourTicket, a 3-message protocol that realizes an offline probabilistic payment between a sender (customer) and receiver (merchant); see Figure 4 for details. For simplicity, we only discuss enforcement of the worst-case payment rate bounds; enforcement of the average-case bound is achieved via essentially the same ideas. Recall that the worst-case bound limits the number of macropayments that occur in a particular time window tw.

1st message (sender ← receiver). The first message of the protocol is from the receiver to the sender and consists of the receiver's *session identifier* sid, *session public key* spk, the list of deposits $\mathcal{D}$ that have been blacklisted in this epoch, and the desired public value v_{pub}. These are constructed as follows. Suppose that the receiver has an address key pair $(\mathsf{apk}_{\mathbf{c}}, \mathsf{ask}_{\mathbf{c}})$ and wishes to receive payments at this address with payment probability p_{r} and macropayment value V_{r}; moreover, suppose that the receiver's per-deposit maximum cumulative average-case payment value rate is $\mathfrak{a}_{\mathsf{r}}$. Then the receiver constructs his session identifier as $\mathsf{sid} := (\mathsf{apk}_{\mathbf{c}}, \mathsf{tw}_{\mathsf{r}})$. To construct the session public key, the receiver samples a new key pair $(\mathsf{pk}_{\mathsf{SIG}}, \mathsf{sk}_{\mathsf{SIG}})$ for the one-time signature scheme, and a new key pair $(\mathsf{pk}_{\mathsf{FMT}}, \mathsf{sk}_{\mathsf{FMT}})$ for the fractional message transfer scheme and sets $\mathsf{spk} := (\mathsf{pk}_{\mathsf{FMT}}, \mathsf{pk}_{\mathsf{SIG}})$. Finally, the deposit blacklist $\mathcal{D}$ consists of the identifiers of deposits seen in punish transactions within the *current* epoch.

2nd message (sender → receiver). The sender now pours his ticket $\mathbf{t}$ into a new coin $\mathbf{c}$ using DAP.Pour, and then uses fractional message transfer to probabilistically transmit the new coin $\mathbf{c}$ to the receiver, while also proving, in zero knowledge, that he did so correctly. We now expand on this description, which hides subtle aspects of our construction.

After pouring his ticket $\mathbf{t}$ into a new coin $\mathbf{c}$ (which results in a DAP pour transaction tx_{p}), the sender uses the deposit $\mathbf{d}$ backing $\mathbf{t}$ to generate two crucial quantities: the worst-case *rate limit tag* wrlt and the *double spend tag* dst. The rate limit tag allows the receiver to enforce the payment value rate bounds required by the economic analysis. The double spend tag allows the receiver to extract deposit revocation information if and only if $\mathbf{t}$ is spent in two macropayments.

A natural strategy would be for the sender to send to the receiver, in the clear, the rate limit tag wrlt, and a FMT ciphertext c_{FMT} containing tx_{p} and dst, along with a non-interactive zero knowledge proof that both were generated correctly. However, doing so does not preserve privacy. Indeed, to ensure that the sender cannot double spend the ticket to herself and escape punishment, the ledger needs to check that the double spend tag was generated correctly. This can be done by verifying the NIZK proof, but to do this would require including the FMT ciphertext, blacklist detection tag, and rate limit tag as part of the NP instance being verified. This is problematic, since *publishing these leaks information about the transfer probability p_{r} and the deposit, both of which are private information.*

To fix this problem, the sender hides wrlt and c_{FMT} inside two commitments ω_0 and ω_1, and then computes a proof of correctness relative to these commitments. More precisely, the first commitment ω_0 hides $m_0 := (\mathsf{sid}, \mathsf{spk}, v_{\mathsf{pub}}, c_{\mathsf{FMT}})$, where sid, spk, and v_{pub} are the receiver's session identifier, session public key, and public value respectively, and c_{FMT} is a FMT ciphertext. The FMT ciphertext c_{FMT}, as before, contains wrlt, tx_{p} and dst, but now also contains randomness r_1 that opens the second commitment ω_1, which in turn hides $m_1 := (\mathsf{tx}_{\mathsf{p}}, \mathsf{dst}, \mathsf{wrlt})$. Thus opening the FMT ciphertext allows the receiver to open ω_1 and obtain the correct $\mathsf{tx}_{\mathsf{p}}, \mathsf{dst}, \mathsf{wrlt}$. Next, the sender generates a non-interactive zero knowledge proof of knowledge π_{pt} asserting that he performed all these steps correctly (see Section 7.2.7). The NIZK also asserts (a) that the deposit $\mathbf{d}$'s receiver address set $\mathcal{R}$ contains the receiver's address public key apk, (b) that $\mathbf{d}$'s identifier has not appeared in punish transactions in the current epoch, and (c) that $\mathbf{d}$'s serial number has not appeared on the ledger prior to the current epoch (that is, $\mathbf{d}$ was not revoked or withdrawn in prior epochs).

Finally, he sends $(\omega_0, m_0, \omega_1, \pi_{\mathsf{pt}})$ and randomness r_0 for opening ω_0 to the receiver. Since the proof is now computed relative to ω_0 and ω_1, and not c_{FMT} and wrlt, it can safely be published to the ledger.

3rd message (sender ← receiver). The receiver uses r_0 and m_0 to open ω_0 and checks that the committed sid, spk and v_{pub} are indeed the correct ones (which were sent in the first message). Next, he checks the correctness of π_{pt}, and finally, using the rate limit tag, he checks that the payment value rate $\mathfrak{a}_{\mathsf{r}}$ has not been exceeded. If these checks pass, he tries to open the FMT ciphertext c_{FMT} inside ω_0. If he is able to successfully open it, he can open ω_1 to obtain tx_{p} and dst. If the ticket $\mathbf{t}$ has already been spent (i.e., the deposit $\mathbf{d}$ has been blacklisted), the receiver recovers the deposit and creates a punish transaction $\mathsf{tx}_{\mathsf{pun}}$. If not, he posts tx_{p} to obtain his payment. Finally, he sends to the sender the secret key $\mathsf{sk}_{\mathsf{FMT}}$ used for decryption, and m', which is the outcome of decryption, to communicate whether the outcome was 'macropayment' or 'nullpayment'.

Outcome verification. Upon receiving the FMT secret key, the sender checks that the FMT ciphertext c_{FMT} decrypts to claimed message m' under the key $\mathsf{sk}_{\mathsf{FMT}}$; this reveals whether the claimed outcome was the correct one. If the receiver sends an incorrect secret key, or does not send anything at all, the sender refreshes his ticket, thereby generating a new ticket $\mathbf{t}'$ and a refresh transaction $\mathsf{tx}_{\mathsf{ref}}$.

7.1.3 Security considerations

We give an intuitive justification of why the probabilistic payment protocol is secure.

Sender security. The fractional hiding property of the FMT scheme ensures that the receiver can only open c_{FMT} with probability p_{r}. Since the commitment ω_1 is hiding and the proof π_{pt} is zero knowledge, the rest of the sender's message is indistinguishable from random. Finally, the security of the "outcome verification" step is guaranteed by the fractional hiding property of the FMT scheme; if the receiver could generate two different secret keys that can decrypt the same FMT ciphertext to different messages, then he could bias the probability of opening the ciphertext in his favor, thus breaking fractional hiding.

The above ensures "intra-protocol" sender security. Post-protocol security requires that the receiver cannot compromise the honest sender's anonymity or cause monetary loss by aborting. This is achieved by allowing the sender to refresh tickets by pouring them into new ones. This breaks the link between the ticket that the receiver has seen and the ticket that the sender can now spend, enabling the sender to freely spend his new ticket.

Receiver security. Opening ω_0 allows the receiver to check that the sender generated the rate limit tags relative to the true session identifier and public key. The fractional binding property of the FMT scheme ensures that the sender cannot alter the probability of opening c_{FMT}. The NIZK proof ensures the correctness of each step.

The above ensures "intra-protocol" receiver security. Achieving post-protocol security is trickier, since we need to ensure that the sender can only create double spend tags that are consistent across independent pour-ticket transactions. In our construction, the sender can attempt to bypass this requirement by manipulating the three inputs that create a double spend tag: the randomness x used for generating the tag, and the deposit $\mathbf{d}$ that is hidden in the tag, and the ticket $\mathbf{t}$ that $\mathbf{d}$ backs.

Preventing reuse of randomness. To prevent recovery of the deposit serial number from multiple double spend tags, the sender could attempt to reuse randomness across each tag. This would prevent recovery, since each receiver would possess the same tag. To prevent this, our construction of a double spend tag dst uses a special one-time signature public key $\mathsf{pk}_{\mathsf{SIG}}$ as randomness. Later, upon receiving the tag, the receiver signs the tag (among other

things) with the secret key $\mathsf{sk}_{\mathsf{SIG}}$ corresponding to $\mathsf{pk}_{\mathsf{SIG}}$. To create two different pour-ticket transactions with the same double spend tag (one to an honest receiver and one back to himself), the sender would thus have to forge a signature, which is computationally infeasible by the security of the signature scheme.

Ensuring $\mathbf{d}$ *backs* $\mathbf{t}$. The NIZK proof created by the sender ensures that the deposit $\mathbf{d}$ hidden in the double spend tag is the one backing $\mathbf{t}$.

'Identical' tickets backed by different deposits. In principle, one could construct two tickets $\mathbf{t}, \mathbf{t}'$ that have the same serial number (and are thus indistinguishable from the point of view of double spending), but are backed by different deposits. Since $\mathbf{t}$ and $\mathbf{t}'$ would share serial numbers, only one of the two could be successfully spent. This could lead to the following attack: the sender generates two such tickets, and pays himself with one, and pays a receiver with the other. When a macropayment occurs, he front runs the receiver to get his self-payment onto the ledger first. The receiver is then robbed of his payment, but also cannot punish the sender, since the double spend tags hide different deposits, making revocation impossible.

However, our construction prevents such an attack by ensuring that the serial number of a note is derived (in part) from its secret information string sec. This property is guaranteed by the DAP scheme (see Sections 5.1 and 5.4).[8]

7.1.4 Regulating type transitions when pouring

The definition of a DAM scheme restricts fund transfers between different note types: coins can be poured into coins, deposits, or tickets; a deposit can be poured into a coin; a ticket can be poured into a coin or ticket. Moreover, some type transitions are handled differently from others: for example, pouring from a set of coins yields a pour-coin transaction that is immediately valid, while pouring from a ticket to a ticket yields a refresh transaction that only becomes valid after a waiting period (the activation delay). We realize most of these fund transfers via DAP pours, but we must also somehow meet the aforementioned restrictions.

The first obstacle is that a note's type is not necessarily known, because we store the type of note in its public information string pub, which is not revealed by a DAP pour transaction. But remember that a DAP scheme allows us to choose, at parameter setup time, a pour predicate that regulates all pour transactions. We thus engineer a pour predicate Π^*_{p}, tailored for our application, that (i) allows only the aforementioned type transitions, and (ii) ensures that the information string info in a DAP pour transaction correctly exposes the type of the note from which we are pouring; see Section 7.2.2 for more details. (It turns out that explicitly exposing the type of output coins is not needed.)

7.2 Building blocks

In addition to DAP schemes (Section 5) and FMT schemes (Section 4), our construction uses standard cryptographic tools (Section 7.2.1), as well as several DAM-specific building blocks: the 'DAM pour predicate' (Section 7.2.2), means of enforcing payment value rate limits (Section 7.2.3), detecting recently revoked deposits (Section 7.2.4), secret sharing for double spend tags (Section 7.2.5), receiver set Merkle trees (Section 7.2.6), and non-interactive zero knowledge proofs of knowledge for a 'pour-ticket' NP relation (Section 7.2.7).

7.2.1 Standard cryptographic tools

Collision-resistant hash functions. A collision-resistant hash function CRH is a function for which it is computationally infeasible to find two inputs x, y such that $x \neq y$ but $\mathsf{CRH}(x) = \mathsf{CRH}(y)$. We use CRH in our construction of double spend tags; see Section 7.2.5. (More precisely, CRH is a *family* indexed by public parameters; for notational simplicity, we do not make these explicit in our construction.)

Pseudorandom functions. A pseudorandom function PRF is a function family, indexed by a random *seed*, whose outputs are computationally indistinguishable from random; we denote by $\mathsf{PRF}_s(x)$ the output of the function when the seed is s, and the input is x. We also assume that PRF is collision resistant, i.e., it is computationally infeasible to find $(s, x) \neq (s', x')$ such that $\mathsf{PRF}_s(x) = \mathsf{PRF}_{s'}(x')$. We use PRF to generate rate limit tags (see Section 7.3).

[8] Note that the zero knowledge proof in the construction cannot prevent such an attack because it has no knowledge of other tickets.

Commitment schemes. A (non-interactive) commitment scheme COMM enables a party to generate a (statistically) hiding and (computationally) binding commitment to a given message. Namely, for every two messages x, y the distributions $\mathsf{COMM}_r(x)$ and $\mathsf{COMM}_r(y)$ are statistically close for random r; moreover, it is computationally infeasible to find two messages x, y and randomness r, r' such that $x \neq y$ but $\mathsf{COMM}_r(x) = \mathsf{COMM}_{r'}(y)$. (More precisely, COMM is a *family* indexed by public parameters; for notational simplicity, we do not make these explicit in our construction.)

One-time signature schemes. A one-time signature scheme is a signature scheme $\mathsf{SIG} := (\mathsf{SIG.Setup}, \mathsf{SIG.Keygen}, \mathsf{SIG.Sign}, \mathsf{SIG.Verify})$ that guarantees message unforgeability only if a key pair is used once. The algorithms have the following syntax: (i) on input a security parameter, $\mathsf{SIG.Setup}$ samples public parameters $\mathsf{pp}_{\mathsf{SIG}}$; (ii) on input $\mathsf{pp}_{\mathsf{SIG}}$, $\mathsf{SIG.Keygen}$ outputs a key pair $(\mathsf{pk}_{\mathsf{SIG}}, \mathsf{sk}_{\mathsf{SIG}})$; (iii) on input public parameters $\mathsf{pp}_{\mathsf{SIG}}$, a secret key $\mathsf{sk}_{\mathsf{SIG}}$, and a message x, $\mathsf{SIG.Sign}$ outputs a signature σ for x; (iv) on input public parameters $\mathsf{pp}_{\mathsf{SIG}}$, a public key $\mathsf{pk}_{\mathsf{SIG}}$, a message x, and a purported signature σ, $\mathsf{SIG.Verify}$ outputs a bit denoting whether σ is a valid signature for x under public key $\mathsf{pk}_{\mathsf{SIG}}$.

7.2.2 Choice of mint and pour predicate

Recall that $\mathsf{DAP.Setup}$ takes as input two predicates: a mint predicate Π_{m} that regulates the minting of coins, and a pour predicate Π_{p} that regulates the pouring of coins. In our DAM scheme construction, we leverage both a mint predicate $\Pi_{\mathsf{m}} := \Pi^*_{\mathsf{m}}$ and a pour predicate $\Pi_{\mathsf{p}} := \Pi^*_{\mathsf{p}}$.

- *Mint predicate.* The mint predicate Π^*_{m} regulates minting of tickets, and ensures that the public key of a ticket is the same as the public key of the underlying deposit.
- *Pour predicate.* The pour predicate Π^*_{p} (a) allows only certain type transitions among coins, deposits, and tickets; and (b) enforces the correct activation delays on these type transitions. To check type transitions, we store a note's type (coin, deposit, or ticket) in the note's public information string, and then Π^*_{p} allows only the following type conversions: coins can be poured into coins, deposits, or tickets; deposits can be poured into coins; tickets can be poured into coins or tickets. Additionally, when pouring coins or tickets to tickets (i.e., via $\mathsf{DAM.PourCoinToTicket}$ or $\mathsf{DAM.RefreshTicket}$), Π^*_{p} enforces that the public keys of input and output notes must *all* be equal. Depending on the type transition, Π^*_{p} enforces the correct activation delay: zero for all, except Δ_{r} when refreshing a ticket (i.e., ticket to ticket) and Δ_{w} when withdrawing a deposit (i.e., deposit to coin). In detail, we define Π^*_{p} as follows (dropping unused inputs):

$\Pi^*_{\mathsf{m}}\left(\begin{array}{c}\text{ledger time } \mathbf{L}.\mathsf{Len},\\ \text{public key } \mathsf{apk},\\ \text{public information } \mathsf{pub},\\ \text{secret information } \mathsf{sec}\end{array}\right)$	$\Pi^*_{\mathsf{p}}\left(\begin{array}{c}\text{ledger time } \mathbf{L}.\mathsf{Len},\\ \text{old input notes } [\mathbf{n}_i]_1^m,\\ \text{new public keys } [\mathsf{apk}_j]_1^n,\\ \text{new public information } [\mathsf{pub}_j]_1^n,\\ \text{transaction information string } \mathsf{info},\\ \text{activation delay } \Delta\end{array}\right)$
1. If $\mathsf{pub} = \mathtt{tk}$: check that $\mathbf{d}.\mathsf{apk} = \mathsf{apk}$.	1. Parse info as $(\mathsf{info}', \mathsf{tp})$ and check that $\mathsf{tp} \in \{\mathtt{cn}, \mathtt{dp}, \mathtt{tk}\}$. 2. For $i \in \{1, \ldots, m\}$: check that $\mathbf{n}_i.\mathsf{pub} = \mathsf{tp}$. 3. If $\mathsf{tp} = \mathtt{cn}$: (a) check that there exists $\mathsf{tp}' \in \{\mathtt{cn}, \mathtt{dp}, \mathtt{tk}\}$ such that $\mathsf{pub}_j = \mathsf{tp}'$ (for all j) and $\Delta = 0$. (b) if $\mathsf{pub}_j = \mathtt{tk}$: check that for all $i \in \{1, \ldots, m\}$ and $j \in \{1, \ldots, n\}$: i. $\mathbf{n}_i.\mathsf{apk} = \mathsf{apk}_j$ ii. $\mathbf{d}_j.\mathsf{apk} = \mathsf{apk}_j$. 4. If $\mathsf{tp} = \mathtt{dp}$: (a) check that $m = n = 1$, $\mathsf{pub}_1 = \mathtt{cn}$, and $\Delta = \Delta_{\mathsf{w}}$. 5. If $\mathsf{tp} = \mathtt{tk}$: (a) check that $m = n = 1$ and $\mathsf{pub}_1 \in \{\mathtt{cn}, \mathtt{tk}\}$; (b) if $\mathsf{pub}_1 = \mathtt{cn}$: check that $\Delta = 0$; (c) if $\mathsf{pub}_1 = \mathtt{tk}$: check that $\Delta = \Delta_{\mathsf{r}}$ and that $\mathbf{n}_1.\mathsf{apk} = \mathbf{n}_1.\mathsf{sec}.\mathsf{apk} = \mathsf{apk}_1$.

7.2.3 Enforcing payment value rates

The deposit value bounds in the economic analysis of Section 3 require receivers to *enforce payment value rates* within a particular time window. To prove that he has not exceeded the bounds on these rates, the sender constructs a *rate limit*

tag and sends it over to the receiver. The receiver maintains a *payment counter* that allows him to use the received rate limit tag to check whether or not the payment value rate has been exceeded.

Rate limit tags. Rate limit tags enable enforcement of payment value rate bounds while simultaneously ensuring privacy and unlinkability of payments backed by the same deposit. A sender computes a rate limit tag by evaluating a PRF seeded by a deposit's secret key $\mathsf{ask}_{\mathbf{d}}$ on a non-repeating counter ctr, the merchant's session identifier sid, and the current ledger epoch $\mathbf{L}.\mathsf{Epoch}$.

Our construction uses two different kinds of rate limit tags: one to track the number of probabilistic payments a deposit $\mathbf{d}$ has backed, and one to track the number macropayments $\mathbf{d}$ has backed. To prevent collisions between the two (which can leak information about the origin of a probabilistic transaction), we compute the two with different PRFs. We wrap up the computation of rate limit tags into specialized interfaces detailed below.

ARLT.CreateTag	WRLT.CreateTag
deposit secret key $\mathsf{ask}_{\mathbf{d}}$ session identifier sid current ledger epoch $\mathbf{L}.\mathsf{Epoch}$	
average-case counter actr	worst-case counter wctr
average-case rate limit tag arlt	worst-case rate limit tag wrlt
1. $\mathsf{arlt} \leftarrow \mathsf{PRF}'_{\mathsf{ask}_{\mathbf{d}}}(\mathsf{sid} \Vert \mathbf{L}.\mathsf{Epoch} \Vert \mathsf{wctr})$. 2. Output arlt.	1. $\mathsf{wrlt} \leftarrow \mathsf{PRF}''_{\mathsf{ask}_{\mathbf{d}}}(\mathsf{sid} \Vert \mathbf{L}.\mathsf{Epoch} \Vert \mathsf{actr})$. 2. Output wrlt.

Payment counters. The payment counter $\mathcal{P}$ is a set data structure that has two associated algorithms:

- $\mathsf{PC.Add}(\mathcal{P}, \mathsf{rlt}, f) \to \mathcal{P}'$.
  This algorithm takes as input a payment counter $\mathcal{P}$, a rate limit tag rlt and a flag $f \in \{\mathtt{A}, \mathtt{W}\}$ that decides whether $\mathcal{P}$ should count the input tag towards the average-case or worst-case bounds, and outputs an updated payment counter $\mathcal{P}'$ that also contains rlt.
- $\mathsf{PC.CheckCounter}(\mathcal{P}, \mathsf{rlt}, f) \to b$.
  This algorithm takes as input a payment counter $\mathcal{P}$, a rate limit tag rlt and a flag $f \in \{\mathtt{A}, \mathtt{W}\}$ (to decide whether $\mathcal{P}$ should check the average-case or worst-case rate limits) and outputs a bit b such that $b = 1$ if and only if rlt is already in $\mathcal{P}$ (our construction of rate limit tags ensures that such a collision can happen if and only if the sender exceeds the rate limit set by the merchant).[9]

7.2.4 Detecting revoked deposits

The deposit value bounds in the economic analysis of Section 3 require receivers to detect and reject payments backed by deposits that have been *blacklisted in the current epoch just before the latest time window*. To enable this while simultaneously hiding information about the deposit, the receiver maintains a *deposit blacklist* that is reset every epoch. When a sender wishes to initiate a probabilistic payment, the receiver sends over this (short) list, and the sender proves that his deposit is not in the blacklist (see Section 7.2.7 for details of this proof).

Deposit blacklist. The deposit blacklist $\mathcal{D}$ is a set data structure containing all the deposit identifiers seen in punish transactions published in the current epoch. It has the following interface:

Updating blacklists: $\mathsf{DB.Add}(\mathcal{D}, \mathsf{id}) \to \mathcal{D}'$.

This algorithm takes as input a deposit blacklist $\mathcal{D}$, a deposit's identifier id and outputs a new deposit blacklist $\mathcal{D}' := \mathcal{D} \cup \{\mathsf{id}\}$.

Committing to a blacklist: $\mathsf{DB.Comm}(\mathcal{D}) \to \gamma_{\mathcal{D}}$.

This algorithm takes as input a deposit blacklist $\mathcal{D}$ and outputs a commitment $\gamma_{\mathcal{D}}$ to $\mathcal{D}$ that allows later proofs of non-membership in $\mathcal{D}$.

[9]Note that even if the sender is dishonest and double spends, the payment counter does not compromise anonymity, since the rate limit tag is pseudorandom to the receiver. Furthermore, if the list of tags is sorted, finding collisions can be done efficiently via binary search.

Proving non-membership: $\mathsf{DB.Prove}(\mathcal{D}, \mathsf{id}) \to \pi_{\mathcal{D}}$.

This algorithm takes as input a deposit blacklist $\mathcal{D}$ and a deposit identifier id, and outputs a proof $\pi_{\mathcal{D}}$ that $\mathcal{D}$ does not contain id.

Verifying non-membership: $\mathsf{DB.Verify}(\gamma_{\mathcal{D}}, \mathsf{id}, \pi_{\mathcal{D}}) \to b$.

This algorithm takes as input a deposit blacklist commitment $\gamma_{\mathcal{D}}$, a deposit identifier id, and a proof $\pi_{\mathcal{D}}$, and outputs a bit b denoting whether or not $\pi_{\mathcal{D}}$ is a valid proof of non-membership relative to $\gamma_{\mathcal{D}}$.

7.2.5 Creating double spend tags and recovering deposits

When a ticket $\mathbf{t}$ participates in a probabilistic payment that results in a macropayment, the receiver obtains, and appends to the ledger, a pour-ticket transaction $\mathsf{tx}_{\mathsf{pt}}$. We include in $\mathsf{tx}_{\mathsf{pt}}$ a *double spend tag* dst, whose purpose is to allow revocation of $\mathbf{t}$'s deposit $\mathbf{d}$ if $\mathbf{t}$ is ever involved in another probabilistic payment that results in a macropayment (which is a detectable double spend). We thus set the double spend tag dst to be a 2-out-of-n secret share [Sha79] of the deposit $\mathbf{d}$'s serial number sn and identifier id, so that any single double spend tag dst in a pour-ticket transaction $\mathsf{tx}_{\mathsf{pt}}$ does not leak information about $\mathbf{d}$, but together with an additional double spend tag dst' in another pour-ticket transaction $\mathsf{tx}'_{\mathsf{pt}}$ (relative to the same ticket) allows anyone to recover sn and id and blacklist the deposit. (Our technique is similar to that used in [Bra93, Fer93] to obtain *traceability after double spends*.)

For notational simplicity, we "wrap" the secret-sharing functionality inside a double spend tag interface that consists of two algorithms $(\mathsf{DST.CreateTag}, \mathsf{DST.GetSecret})$ for the two tasks of creating a double spend tag and of recovering a deposit from two pour-ticket transactions relative to the same (double-spent) ticket, as follows.

- $\mathsf{DST.CreateTag}$ takes as input a ticket $\mathbf{t}$, the $\mathbf{t}$'s address secret key $\mathsf{ask}_{\mathbf{t}}$, a deposit $\mathbf{d}$, $\mathbf{d}$'s address secret key $\mathsf{ask}_{\mathbf{d}}$, and a session public key spk (of the probabilistic payment), and pseudorandomly constructs the double spend tag as follows: retrieve from spk the one-time signature public key $\mathsf{pk}_{\mathsf{SIG}}$; compute sn and id from $\mathbf{d}$ and $\mathsf{ask}_{\mathbf{d}}$; construct the line ℓ that has slope $\mathsf{PRF}_{\mathsf{ask}_{\mathbf{t}}}(\mathsf{GetID}(\mathbf{t}, \mathsf{ask}_{\mathbf{t}}))$ and evaluates to $\mathsf{sn}\|\mathsf{id}$ at zero; set dst to be the value of ℓ at $\mathsf{CRH}(\mathsf{pk}_{\mathsf{SIG}})$. (The use of CRH here is merely to shrink $\mathsf{pk}_{\mathsf{SIG}}$ to fit in an element of the underlying implicit finite field, if needed.)
- $\mathsf{DST.GetSecret}$ takes as input two pour-ticket transactions $\mathsf{tx}_{\mathsf{pt}}, \mathsf{tx}'_{\mathsf{pt}}$, and recovers the serial number sn and identifier id of the deposit $\mathbf{d}$ that backed both payments as follows: retrieve from $\mathsf{tx}_{\mathsf{pt}}$ a one-time signature public key $\mathsf{pk}_{\mathsf{SIG}}$ and double spend tag dst and similarly retrieve $\mathsf{pk}'_{\mathsf{SIG}}$ and dst' from $\mathsf{tx}'_{\mathsf{pt}}$; recover the line $\ell(X)$ that passes through the points $(x, y) := (\mathsf{CRH}(\mathsf{pk}_{\mathsf{SIG}}), \mathsf{dst})$ and $(x', y') := (\mathsf{CRH}(\mathsf{pk}'_{\mathsf{SIG}}), \mathsf{dst}')$; set $(\mathsf{sn}\|\mathsf{id})$ to be the value of ℓ at zero.

Selecting the line ℓ depending on the ticket identifier ensures that different probabilistic payments that use the same ticket will refer to the same line. (It is computationally infeasible to mint different tickets that share the same secret key and, moreover, the deposit $\mathbf{d}$ is computationally bound to the ticket at ticket minting time.) Furthermore, each probabilistic payment evaluates ℓ at a different point by using different one-time public keys (chosen by the receiver), to ensure that a malicious sender does not reveal the same share in two different macropayments (which would prevent recovery of ℓ from those two macropayments). Note that we are forced to use $\mathsf{CRH}(\mathsf{pk}_{\mathsf{SIG}})$ as the evaluation point, rather than simply using $\mathsf{CRH}(\mathsf{spk}) = \mathsf{CRH}(\mathsf{pk}_{\mathsf{FMT}}, \mathsf{pk}_{\mathsf{SIG}})$, because others would need to know $\mathsf{CRH}(\mathsf{spk})$ to recover the hidden secret, and this may reveal information about $\mathsf{pk}_{\mathsf{FMT}}$, and thus also about the payment probability p_{r}, thereby revealing private information about the payment. The pseudocode below summarizes the construction described above.

DST.CreateTag	DST.GetSecret
ticket $\mathbf{t}$ ticket address secret key $\mathsf{ask}_{\mathbf{t}}$ deposit $\mathbf{d}$ deposit address secret key $\mathsf{ask}_{\mathbf{d}}$ session public key spk	pour-ticket transactions $\mathsf{tx}_{\mathsf{pt}}$ and $\mathsf{tx}'_{\mathsf{pt}}$
double spend tag dst	deposit serial number sn and identifier id
1. Compute $\mathsf{sn} \leftarrow \mathsf{GetSN}(\mathbf{d}, \mathsf{ask}_{\mathbf{d}})$; $\mathsf{id} \leftarrow \mathsf{GetID}(\mathbf{d}, \mathsf{ask}_{\mathbf{d}})$. 2. Set secret $s := \mathsf{sn}\|\mathsf{id}$. 3. Set point $t := \mathsf{CRH}(\mathsf{spk}.\mathsf{pk}_{\mathsf{SIG}})$. 4. Set slope $m := \mathsf{PRF}_{\mathsf{ask}_{\mathbf{t}}}(\mathsf{GetID}(\mathbf{t}, \mathsf{ask}_{\mathbf{t}}))$. 5. Construct line $\ell(X) := m \cdot X + s$. 6. Compute $\mathsf{dst} := \ell(t) = m \cdot t + s$. 7. Output dst.	1. Set $x := \mathsf{CRH}(\mathsf{pk}_{\mathsf{SIG}})$, where $\mathsf{pk}_{\mathsf{SIG}}$ is the one-time signature public key in $\mathsf{tx}_{\mathsf{pt}}$. 2. Set $y := \mathsf{dst}$, where dst is the double spend tag in $\mathsf{tx}_{\mathsf{pt}}$. 3. Set $x' := \mathsf{CRH}(\mathsf{pk}'_{\mathsf{SIG}})$, where $\mathsf{pk}'_{\mathsf{SIG}}$ is the one-time signature public key in $\mathsf{tx}'_{\mathsf{pt}}$. 4. Set $y' := \mathsf{dst}'$, where dst' is the double spend tag in $\mathsf{tx}'_{\mathsf{pt}}$. 5. Interpolate points (x, y) and (x', y') to recover the line ℓ. 6. Compute $\mathsf{sn}\|\mathsf{id} := \ell(0)$. 7. Output $(\mathsf{sn}, \mathsf{id})$.

7.2.6 Receiver address set Merkle tree

Recall that to successfully enforce payment rates, we associate with each deposit a "receiver address set". The deposit's value then needs to be large enough to support transactions with the receivers in this set. Thus, to pay a receiver having address public key apk with a ticket $\mathbf{t}$, a sender must prove that apk is in the receiver address set $\mathcal{R}$ of $\mathbf{t}$'s deposit, and that the value of this deposit is greater than (1) $\sum_{\mathsf{r}\in\mathcal{R}} \mathfrak{w}_{\mathsf{r}}$ for the worst-case bound; and (2) $(\sum_{\mathsf{r}\in\mathcal{R}} \mathfrak{a}_{\mathsf{r}}) + \max_{\mathsf{r}\in\mathcal{R}} V_{\mathsf{r}}$ for the average-case bound.

To enable efficient proofs of these, we construct a *monotonic Merkle tree* over the receiver address set. Such a tree not only enables efficient proofs of membership, but also enables efficient proofs of the fact that for any tree, the value at the root is the true aggregate of those elements in the set for which a valid path exists in the tree. For concreteness, below we discuss these properties with respect to summation; obtaining similar guarantees for other aggregation operations is straightforward. Such *summation Merkle trees* have been considered before (see [Wil14] and [NBF+16, pg. 93]).

A *summation Merkle tree* is constructed as follows: the leaves consist of elements of a set, and each internal node stores the sum $s = v_l + v_r$ of the values v_l, v_r of its children, and also stores a hash $h = \mathsf{CRH}(\mathsf{left}\|\mathsf{right}\|s)$ of the children and their sum. Aside from providing an efficient set membership proof (like standard Merkle trees), a such a tree also provides the following guarantee: if there are n valid authentication paths in the tree (for leaf values $v_1, \ldots, v_n$), then the sum at the root is at least $\sum_{i=1}^{n} v_i$.[10] This guarantee is sufficient for our needs, because our economic analysis only requires the deposit to be as large as the cumulative payment rates for the merchants a sender transacts with. If a path in the tree is not valid, the path verification step in the zero knowledge proof (see Section 7.2.7) will fail, causing the receiver to terminate the transaction. Thus the sender will only be able to transact with those receivers for whom he can produce a valid authentication path.

The above idea can be extended in a straightforward manner to support other aggregation operations like $\max$, $\min$, multiplication, etc., with suitably modified notions of comparison. We use such monotonic Merkle trees to enable the proofs required for successful probabilistic payment, wrapping up calls to the data structure into a *Receiver Set Tree* (RST) interface:

1. *Set commitment.* $\mathsf{RST.Comm}(\mathcal{R}) \to \alpha_{\mathcal{R}}$. On input a receiver address set $\mathcal{R}$, RST.Comm generates a short commitment $\alpha_{\mathcal{R}}$ to the set.

2. *Proving membership and summation.* $\mathsf{RST.Prove}(\mathcal{R}, \mathsf{apk}) \to \pi_{\mathcal{R}}$. On input a receiver address set $\mathcal{R}$ and an address public key apk, RST.Prove outputs a proof $\pi_{\mathcal{R}}$ of the following.

   (a) *Valid receiver address:* $\mathsf{apk} \in \mathcal{R}$.

[10] This is easy to verify: we proceed via induction on n. The case for $n = 1$ is trivial. Assume that the hypothesis is true for $n = k$. Then we prove that if the $k + 1^{th}$ path is valid, then the sum at the root is $\sum_{i=1}^{k+1} l_i$.

Let the $k + 1^{th}$ path intersect a previous path p_i at an internal vertex v, and let the values of the two paths at v be different. Since both paths are valid, the hashes at v must be equal. But these two facts cannot be reconciled unless the collision resistance of the hash function is broken. Since we assume collision resistance, this means that the values of v in both p_i and p_{k+1} must be equal. This means that $v \geq l_i + l_{k+1}$. Propagating this sum up the tree, we obtain that the sum at the root must be at least as large as $\sum_{i=1}^{k+1} l_i$.

(b) *Worst-case bound:* $\alpha_{\mathcal{R}}.W \geq \mathsf{apk}.\mathfrak{w}_{\mathsf{r}}$, where W is the sum of the worst-case bounds in $\mathcal{R}$.
(c) $\alpha_{\mathcal{R}}.A + \alpha_{\mathcal{R}}.V_{\mathsf{max}} \geq \mathsf{apk}.\mathfrak{a}_{\mathsf{r}} + \mathsf{apk}.V_{\mathsf{r}}$, where A is the sum of the average-case bounds in $\mathcal{R}$, and the V_{max} is the maximum transaction value among all the addresses in $\mathcal{R}$.

Proofs for these conditions can be obtained easily from the aforementioned *monotonic Merkle trees*.

3. *Verifying proofs.* $\mathsf{RST.Verify}(\alpha_{\mathcal{R}}, \pi_{\mathcal{R}}) \to b$. On input a commitment to a receiver address set $\alpha_{\mathcal{R}}$ and a (claimed) proof of membership and deposit size $\pi_{\mathcal{R}}$, RST.Verify outputs a bit b denoting if $\pi_{\mathcal{R}}$ is a valid proof relative to $\alpha_{\mathcal{R}}$.

7.2.7 The pour-ticket NP relation

We use non-interactive zero knowledge proofs of knowledge to enable a sender (customer) to prove to a receiver (merchant) that he has committed to well-formed double spend tag, new coin, and pour transaction. More precisely, the proofs have to be *simulation extractable*, i.e., they continue to be proofs of knowledge even given previous simulated proofs [Sah99, DDO+01]. This primitive is a tuple $\mathsf{NIZK} = (\mathsf{NIZK.Setup}, \mathsf{NIZK.Prove}, \mathsf{NIZK.Verify})$ with the following syntax: (i) *(setup)* given a security parameter λ and the specification of an NP relation $\mathcal{R}$, NIZK.Setup outputs a common reference string crs; (ii) *(proving)* given crs and an instance-witness pair $(\mathbb{x}, \mathbb{w}) \in \mathcal{R}$, NIZK.Prove outputs a proof π; (iii) *(verifying)* given crs, instance $\mathbb{x}$, and proof π, NIZK.Verify outputs a decision bit. We use the primitive NIZK *only* for the following NP relation.

The NP relation $\mathcal{R}_{\mathsf{pt}}$. The NP relation $\mathcal{R}_{\mathsf{pt}}$ consists of instance-witness pairs $(\mathbb{x}, \mathbb{w})$ of the form

$$\mathbb{x} := (\mathbf{L}.\mathsf{Len}, \mathbf{L}.\mathsf{SNRoot}, \mathbf{L}.\mathsf{CMRoot}, \mathsf{pk}_{\mathsf{SIG}}, \omega_0, \omega_1)$$

$$\mathbb{w} := \begin{pmatrix} \mathbf{t}, \mathsf{ask}_{\mathbf{t}}, \mathbf{d}, \mathsf{ask}_{\mathbf{d}} & \mathbf{c}, \mathsf{tx}_{\mathsf{p}}, \mathsf{info} & \pi_{\mathcal{R}}, \pi_{\mathsf{sn}}, \pi_{\mathsf{cm}}, \pi_{\mathcal{D}} & c_{\mathsf{FMT}}, r_0, r_1 \\ \mathsf{sid}, \mathsf{spk}, v_{\mathsf{pub}} & \mathsf{dst}, \mathsf{arlt}, \mathsf{wrlt} & \mathsf{actr}, \mathsf{wctr}, \gamma_{\mathcal{D}} & \end{pmatrix}$$

that satisfy the following conditions.

Condition	Explanation
$\mathsf{sid} = (\mathsf{apk}_{\mathbf{c}}, \mathsf{tw}_{\mathsf{r}})$	The session identifier sid has the correct contents.
$\mathsf{spk} = (\mathsf{pk}_{\mathsf{FMT}}, \mathsf{pk}_{\mathsf{SIG}})$	The session public key spk has the correct contents.
$\mathbf{t}.\mathsf{pub} = \mathtt{tk}\ \&\ \mathbf{t}.\mathsf{sec} = \mathbf{d}\ \&\ \mathbf{t}.v = V + v_{\mathsf{pub}}$	The ticket $\mathbf{t}$ is backed by the deposit $\mathbf{d}$ and is of the correct value.
$(\mathbf{d}.\mathsf{apk}, \mathsf{ask}_{\mathbf{d}}) \in \mathsf{DAP.CreateAddr}(\mathsf{pp}.\mathsf{pp}_{\mathsf{DAP}}, \bot)$	The address secret key $\mathsf{ask}_{\mathbf{d}}$ is paired with $\mathbf{d}$'s address public key.
$\mathbf{d}.\mathsf{pub} = \mathtt{dp}$ $\mathsf{RST.Verify}(\mathbf{d}.\mathsf{sec}, \mathsf{apk}_{\mathbf{c}}, \pi_{\mathcal{R}}) = 1$ $\mathbf{L}.\mathsf{CMVerify}(\mathbf{L}.\mathsf{CMRoot}, \mathbf{d}.\mathsf{cm}, \pi_{\mathsf{cm}}) = 1$ $\mathbf{L}.\mathsf{SNVerify}(\mathbf{L}.\mathsf{SNRoot}, \mathsf{GetSN}(\mathbf{d}, \mathsf{ask}_{\mathbf{d}}), 0, \pi_{\mathsf{sn}}) = 1$ $\mathsf{DB.Verify}(\gamma_{\mathcal{D}}, \mathsf{GetID}(\mathbf{d}, \mathsf{ask}_{\mathbf{d}}), \pi_{\mathcal{D}}) = 1$	The deposit $\mathbf{d}$ has large enough value and can be used in a payment to $\mathsf{apk}_{\mathbf{c}}$, was minted, its serial number is not on the serial number list, and its identifier is not in the receiver's deposit blacklist.
$\mathsf{actr} \in \{1, \ldots, \mathfrak{a}_{\mathsf{r}}/p_{\mathsf{r}}V_{\mathsf{r}}\}$ $\mathsf{arlt} = \mathsf{ARLT.CreateTag}(\mathsf{ask}_{\mathbf{d}}, \mathsf{sid}, \mathsf{actr})$	The average-case rate limit tag arlt is computed correctly.
$\mathsf{wctr} \in \{1, \ldots, \mathfrak{w}_{\mathsf{r}}/p_{\mathsf{r}}V_{\mathsf{r}}\}$ $\mathsf{wrlt} = \mathsf{WRLT.CreateTag}(\mathsf{ask}_{\mathbf{d}}, \mathsf{sid}, \mathsf{wctr})$	The worst-case rate limit tag wrlt is computed correctly.
$\mathsf{dst} = \mathsf{DST.CreateTag}(\mathbf{t}, \mathsf{ask}_{\mathbf{t}}, \mathbf{d}, \mathsf{ask}_{\mathbf{d}}, \mathsf{spk})$	The double spend tag dst is computed correctly.
$\mathsf{pub} = \mathtt{cn}\ \&\ \mathsf{sec} = \bot\ \&\ \mathsf{info}' = (\mathsf{info}, \mathtt{tk})$ $(\mathbf{c}, \mathsf{tx}_{\mathsf{p}}) \in \mathsf{DAP.Pour}^{\mathbf{L}}(\mathsf{pp}.\mathsf{pp}_{\mathsf{DAP}}, \mathbf{t}, \mathsf{ask}_{\mathbf{t}}, V_{\mathsf{r}}, \mathsf{apk}_{\mathbf{c}}, \mathsf{pub}, \mathsf{sec}, v_{\mathsf{pub}}, \mathsf{info}', 0)$	The new coin $\mathbf{c}$ and pour transaction tx_{p} are computed correctly.
$c_{\mathsf{FMT}} \in \mathsf{FMT.Encrypt}(\mathsf{pp}_{\mathsf{FMT}}, \mathsf{pk}_{\mathsf{FMT}}, (\mathsf{tx}_{\mathsf{p}}, \mathsf{dst}, \mathsf{wrlt}, r_1))$ $\omega_1 = \mathsf{COMM}_{r_1}(\mathsf{tx}_{\mathsf{p}}, \mathsf{dst}, \mathsf{wrlt})$ $\omega_0 = \mathsf{COMM}_{r_0}(\mathsf{sid}, \mathsf{spk}, v_{\mathsf{pub}}, \gamma_{\mathcal{D}}, c_{\mathsf{FMT}}, \mathsf{arlt})$	The FMT ciphertext and commitments are computed correctly.

7.3 Construction

DAM.Setup	DAM.CreateAddr	DAM.Receive$^{\mathbf{L}}$
security parameter 1^λ —	public parameters pp probabilistic payment specification spec	public parameters pp address key pair $(\mathsf{apk}, \mathsf{ask})$
public parameters pp	address key pair $(\mathsf{apk}, \mathsf{ask})$	set of (unspent) received notes
1. $\mathsf{pp}_{\mathsf{DAP}} \leftarrow \mathsf{DAP.Setup}(1^\lambda, \Pi^*_{\mathsf{m}}, \Pi^*_{\mathsf{p}})$ 2. $\mathsf{pp}_{\mathsf{FMT}} \leftarrow \mathsf{FMT.Setup}(1^\lambda)$ 3. $\mathsf{pp}_{\mathsf{SIG}} \leftarrow \mathsf{SIG.Setup}(1^\lambda)$ 4. $\mathsf{crs}_{\mathsf{pt}} \leftarrow \mathsf{NIZK.Setup}(1^\lambda, \mathcal{R}_{\mathsf{pt}})$ 5. output $\mathsf{pp} := (\mathsf{pp}_{\mathsf{DAP}}, \mathsf{pp}_{\mathsf{FMT}}, \mathsf{pp}_{\mathsf{SIG}}, \mathsf{crs}_{\mathsf{pt}})$	1. $\mathsf{meta} := \mathsf{spec}$ 2. $(\mathsf{apk}, \mathsf{ask}) \leftarrow \mathsf{DAP.CreateAddr}(\mathsf{pp}.\mathsf{pp}_{\mathsf{DAP}}, \mathsf{meta})$ 3. output $(\mathsf{apk}, \mathsf{ask})$	1. output $\mathsf{DAP.Receive}^{\mathbf{L}}(\mathsf{pp}.\mathsf{pp}_{\mathsf{DAP}}, (\mathsf{apk}, \mathsf{ask}))$ (Pour-coin, pour-ticket, withdraw, and refresh transactions can be viewed as DAP pour transactions, and so DAP.Receive can retrieve unspent notes from them.)

DAM.MintCoin$^{\mathbf{L}}$	DAM.MintDeposit$^{\mathbf{L}}$	DAM.MintTicket$^{\mathbf{L}}$
public parameters pp value v address public key apk		
—	receiver address set $\mathcal{R}$	deposit number $\mathbf{d}$
coin $\mathbf{c}$ such that $\mathbf{c}.\mathsf{pub} = \mathtt{cn}$ $\mathbf{c}.\mathsf{sec} = \bot$	deposit $\mathbf{d}$ such that $\mathbf{d}.\mathsf{pub} = \mathtt{dp}$ $\mathbf{d}.\mathsf{sec} =$ a commitment to $\mathcal{R}$	ticket $\mathbf{t}$ such that $\mathbf{t}.\mathsf{pub} = \mathtt{tk}$ $\mathbf{t}.\mathsf{sec} = \mathbf{d}$
mint transaction tx_{m}		
1. $\mathsf{pub} := \mathtt{cn}$ 2. $\mathsf{sec} := \bot$	1. $\mathsf{pub} := \mathtt{dp}$ 2. $\mathsf{sec} := \mathsf{RST.Comm}(\mathcal{R})$	1. $\mathsf{pub} := \mathtt{cn}$ 2. $\mathsf{sec} := \mathbf{d}$
3. $(\mathbf{n}, \mathsf{tx}_{\mathsf{m}}) \leftarrow \mathsf{DAP.Mint}(\mathsf{pp}.\mathsf{pp}_{\mathsf{DAP}}, v, \mathsf{apk}, \mathsf{pub}, \mathsf{sec})$ 4. output $(\mathbf{n}, \mathsf{tx}_{\mathsf{m}})$		

DAM.PourCoinToCoin$^{\mathbf{L}}$	DAM.PourCoinToDeposit$^{\mathbf{L}}$	DAM.PourCoinToTicket$^{\mathbf{L}}$
public parameters pp old coins $[\mathbf{c}_i]_1^m$ old address secret keys $[\mathsf{ask}_i]_1^m$ new note values $[v_j]_1^n$ new address public keys $[\mathsf{apk}_j]_1^n$ public value v_{pub} transaction information string info		
—	receiver address sets $[\mathcal{R}_j]_1^n$	deposits $[\mathbf{d}_j]_1^n$
new coins $[\mathbf{c}_j]_1^n$ such that $\mathbf{c}_j.\mathsf{pub} = \mathtt{cn}$ $\mathbf{c}_j.\mathsf{sec} = \bot$	new deposits $[\mathbf{d}_j]_1^n$ such that $\mathbf{d}_j.\mathsf{pub} = \mathtt{dp}$ $\mathbf{d}_j.\mathsf{sec} =$ a commitment to $\mathcal{R}_j$	new tickets $[\mathbf{t}_j]_1^n$ such that $\mathbf{t}_j.\mathsf{pub} = \mathtt{tk}$ $\mathbf{t}_j.\mathsf{sec} = \mathbf{d}_j$
pour-coin transaction $\mathsf{tx}_{\mathsf{pc}}$		
1. $\mathsf{pub}_j := \mathtt{cn}$ 2. $\mathsf{sec}_j := \bot$	1. $\mathsf{pub}_j := \mathtt{dp}$ 2. $\mathsf{sec}_j := \mathsf{RST.Comm}(\mathcal{R}_j)$	1. $\mathsf{pub}_j := \mathtt{tk}$ 2. $\mathsf{sec}_j := \mathbf{d}_j$
3. $\mathsf{info}' := (\mathsf{info}, \mathtt{cn})$ 4. $([\mathbf{n}_j]_1^n, \mathsf{tx}_{\mathsf{p}}) \leftarrow \mathsf{DAP.Pour}^{\mathbf{L}}(\mathsf{pp}.\mathsf{pp}_{\mathsf{DAP}}, [\mathbf{c}_i]_1^m, [\mathsf{ask}_i]_1^m, [v_j]_1^n, [\mathsf{apk}_j]_1^n, [\mathsf{pub}_j]_1^n, [\mathsf{sec}_j]_1^n, v_{\mathsf{pub}}, \mathsf{info}', 0)$ 5. output $([\mathbf{n}_j]_1^n, \mathsf{tx}_{\mathsf{pc}})$, where $\mathsf{tx}_{\mathsf{pc}} := \mathsf{tx}_{\mathsf{p}}$		

DAM.WithdrawDeposit	DAM.RefreshTicket$^{\mathbf{L}}$	DAM.Punish
public parameters pp old deposit $\mathbf{d}$ old deposit address secret key $\mathsf{ask}_{\mathbf{d}}$ new coin address public key apk public value v_{pub} transaction information string info	public parameters pp old ticket $\mathbf{t}$ old ticket address secret key $\mathsf{ask}_{\mathbf{t}}$ new ticket address public key apk public value v_{pub} transaction information string info	pour-ticket transactions $\mathsf{tx}_{\mathsf{pt}}$ and $\mathsf{tx}'_{\mathsf{pt}}$ current deposit blacklist $\mathcal{D}$
new coin $\mathbf{c}$ withdraw transaction $\mathsf{tx}_{\mathsf{wd}}$	new ticket $\mathbf{t}'$ refresh transaction $\mathsf{tx}_{\mathsf{ref}}$	punish transaction $\mathsf{tx}_{\mathsf{pun}}$ updated deposit blacklist $\mathcal{D}'$
1. $\mathsf{pub} := \mathtt{cn}$ 2. $\mathsf{sec} := \bot$ 3. $v := \mathbf{d}.v - v_{\mathsf{pub}}$ 4. $\mathsf{info}' := (\mathsf{info}, \mathtt{dp})$ 5. $(\mathbf{c}, \mathsf{tx}_{\mathsf{p}}) \leftarrow \mathsf{DAP.Pour}^{\mathbf{L}}(\mathsf{pp}.\mathsf{pp}_{\mathsf{DAP}}, \alpha)$ where $\alpha := (\mathbf{d}, \mathsf{ask}_{\mathbf{d}}, v, \mathsf{apk}, \mathsf{pub}, \mathsf{sec}, v_{\mathsf{pub}}, \mathsf{info}', \Delta_{\mathsf{w}})$ 6. output $(\mathbf{c}, \mathsf{tx}_{\mathsf{wd}})$, where $\mathsf{tx}_{\mathsf{wd}} := \mathsf{tx}_{\mathsf{p}}$	1. $\mathsf{pub} := \mathtt{tk}$ 2. $\mathsf{sec} := \mathbf{t}.\mathbf{d}$ 3. $v := \mathbf{t}.v - v_{\mathsf{pub}}$ 4. $\mathsf{info}' := (\mathsf{info}, \mathtt{tk})$ 5. $(\mathbf{t}', \mathsf{tx}_{\mathsf{p}}) \leftarrow \mathsf{DAP.Pour}^{\mathbf{L}}(\mathsf{pp}.\mathsf{pp}_{\mathsf{DAP}}, \alpha)$ where $\alpha := (\mathbf{t}, \mathsf{ask}_{\mathbf{t}}, v, \mathsf{apk}, \mathsf{pub}, \mathsf{sec}, v_{\mathsf{pub}}, \mathsf{info}', \Delta_{\mathsf{r}})$ 6. output $(\mathbf{t}', \mathsf{tx}_{\mathsf{ref}})$, where $\mathsf{tx}_{\mathsf{ref}} := \mathsf{tx}_{\mathsf{p}}$	1. $(\mathsf{sn}, \mathsf{id}) \leftarrow \mathsf{DST.GetSecret}(\mathsf{tx}_{\mathsf{pt}}, \mathsf{tx}'_{\mathsf{pt}})$ 2. $\mathcal{D}' \leftarrow \mathsf{DB.Add}(\mathcal{D}, \mathsf{id})$ 3. $\mathsf{tx}_{\mathsf{pun}} := (\mathsf{tx}_{\mathsf{pt}}, \mathsf{tx}'_{\mathsf{pt}}, \mathsf{sn}, \mathsf{id})$ 4. output $(\mathsf{tx}_{\mathsf{pun}}, \mathcal{D}')$

Figure 3: Construction of a DAM scheme (part 1 of 3).

DAM.PourTicket$^{\mathbf{L}}$

SENDER	RECEIVER
public parameters pp old ticket $\mathbf{t}$ old ticket address secret key $\mathsf{ask}_{\mathbf{t}}$ deposit $\mathbf{d}$ deposit address secret key $\mathsf{ask}_{\mathbf{d}}$ deposit receiver address set $\mathcal{R}_{\mathbf{d}}$ transaction information string info average-case and worst-case counters actr, wctr	public parameters pp current deposit blacklist $\mathcal{D}$ current payment counter $\mathcal{P}$ new coin address key pair $(\mathsf{apk}_{\mathbf{c}}, \mathsf{ask}_{\mathbf{c}})$ receiver time window tw_{r} public value v_{pub}
$\mathsf{status} \in \{\mathtt{null}, \mathtt{macro}, \mathtt{fail}\}$ if $\mathsf{status} = \mathtt{fail}$: also a new ticket $\mathbf{t}'$ & refresh transaction $\mathsf{tx}_{\mathsf{ref}}$	$\mathsf{status} \in \{\mathtt{null}, \mathtt{macro}, \mathtt{fail}, \mathtt{double}\}$ updated payment counter $\mathcal{P}'$ if $\mathsf{status} = \mathtt{double}$: also an updated blacklist $\mathcal{D}'$ & punish transaction $\mathsf{tx}_{\mathsf{pun}}$ if $\mathsf{status} = \mathtt{macro}$: also a new coin $\mathbf{c}$ & pour-ticket transaction $\mathsf{tx}_{\mathsf{pt}}$

STEP 1: GENERATE ONE-TIME KEY PAIRS.
1. $(\mathsf{pk}_{\mathsf{SIG}}, \mathsf{sk}_{\mathsf{SIG}}) \leftarrow \mathsf{SIG.Keygen}(\mathsf{pp.pp}_{\mathsf{SIG}})$.
2. $(\mathsf{pk}_{\mathsf{FMT}}, \mathsf{sk}_{\mathsf{FMT}}) \leftarrow \mathsf{FMT.Keygen}(\mathsf{pp.pp}_{\mathsf{FMT}}, p_{\mathsf{r}})$.
3. $\gamma_{\mathcal{D}} \leftarrow \mathsf{DB.Comm}(\mathcal{D})$.
4. $\mathsf{sid} := (\mathsf{apk}_{\mathbf{c}}, \mathsf{tw}_{\mathsf{r}})$ and $\mathsf{spk} := (\mathsf{pk}_{\mathsf{FMT}}, \mathsf{pk}_{\mathsf{SIG}})$.

$\mathsf{sid}, \mathsf{spk}, \mathcal{D}, \gamma_{\mathcal{D}}, v_{\mathsf{pub}}$ ⟵ SEND PARAMETERS TO SENDER.

STEP 2: COMPUTE TAGS & POUR TICKET.
1. Parse sid as $(\mathsf{apk}_{\mathbf{c}}, \mathsf{tw}_{\mathsf{r}})$ and spk as $(\mathsf{pk}_{\mathsf{FMT}}, \mathsf{pk}_{\mathsf{SIG}})$.
2. Parse $\mathsf{apk}_{\mathbf{c}}.\mathsf{meta}$ as $(\mathfrak{a}_{\mathsf{r}}, \mathfrak{w}_{\mathsf{r}}, p_{\mathsf{r}}, V_{\mathsf{r}})$.
3. Check that $\mathsf{apk}_{\mathbf{c}}$ is indeed the expected one.
4. $\mathsf{arlt} \leftarrow \mathsf{ARLT.CreateTag}(\mathsf{ask}_{\mathbf{d}}, \mathsf{sid}, \mathsf{actr})$.
5. $\mathsf{wrlt} \leftarrow \mathsf{WRLT.CreateTag}(\mathsf{ask}_{\mathbf{d}}, \mathsf{sid}, \mathsf{wctr})$.
6. $\mathsf{dst} \leftarrow \mathsf{DST.CreateTag}(\mathbf{t}, \mathsf{ask}_{\mathbf{t}}, \mathbf{d}, \mathsf{ask}_{\mathbf{d}}, \mathsf{spk})$.
7. $v := \mathbf{t}.v - v_{\mathsf{pub}}$.
8. $\mathsf{pub} := \mathsf{cn}$ and $\mathsf{sec} := \bot$.
9. $\mathsf{info}' := (\mathsf{info}, \mathtt{tk})$.
10. $(\mathbf{c}, \mathsf{tx}_{\mathsf{p}}) \leftarrow \mathsf{DAP.Pour}^{\mathbf{L}}(\mathsf{pp.pp}_{\mathsf{DAP}}, \mathbf{t}, \mathsf{ask}_{\mathbf{t}}, v, \mathsf{apk}_{\mathbf{c}}, \mathsf{pub}, \mathsf{sec}, v_{\mathsf{pub}}, \mathsf{info}', 0)$.

STEP 3: ENCRYPT & COMMIT.
1. Sample COMM randomness r_0, r_1.
2. $m_1 := (\mathsf{tx}_{\mathsf{p}}, \mathsf{dst}, \mathsf{wrlt})$.
3. $c_{\mathsf{FMT}} \leftarrow \mathsf{FMT.Encrypt}(\mathsf{pp}_{\mathsf{FMT}}, \mathsf{pk}_{\mathsf{FMT}}, m_1 \| r_1)$.
4. $m_0 := (\mathsf{sid}, \mathsf{spk}, v_{\mathsf{pub}}, \gamma_{\mathcal{D}}, c_{\mathsf{FMT}}, \mathsf{arlt})$.
5. $\omega_0 := \mathsf{COMM}_{r_0}(m_0)$ and $\omega_1 := \mathsf{COMM}_{r_1}(m_1)$.

STEP 4: GENERATE PROOF OF CORRECTNESS.
1. $\pi_{\mathcal{R}} \leftarrow \mathsf{RST.Prove}(\mathcal{R}_{\mathbf{d}}, \mathsf{apk}_{\mathbf{c}})$.
2. $\pi_{\mathcal{D}} \leftarrow \mathsf{DB.Prove}(\mathcal{D}, \mathsf{GetID}(\mathbf{d}, \mathsf{ask}_{\mathbf{d}}))$.
3. $\pi_{\mathsf{cm}} \leftarrow \mathbf{L}.\mathsf{CMProve}(\mathbf{d}.\mathsf{cm})$.
4. $(b_{\mathsf{sn}}, \pi_{\mathsf{sn}}) \leftarrow \mathbf{L}.\mathsf{SNProve}(\mathsf{sn})$ where $\mathsf{sn} \leftarrow \mathsf{GetSN}(\mathbf{d}, \mathsf{ask}_{\mathbf{d}})$.
5. $\mathbb{x} := (\mathbf{L}.\mathsf{Len}, \mathbf{L}.\mathsf{SNRoot}, \mathbf{L}.\mathsf{CMRoot}, \mathsf{pk}_{\mathsf{SIG}}, \omega_0, \omega_1)$.
6. $\mathbb{w} := \begin{pmatrix} \mathbf{t}, \mathsf{ask}_{\mathbf{t}}, \mathbf{d}, \mathsf{ask}_{\mathbf{d}} & \mathbf{c}, \mathsf{tx}_{\mathsf{p}}, \mathsf{info} & \pi_{\mathcal{R}}, \pi_{\mathsf{sn}}, \pi_{\mathsf{cm}}, \pi_{\mathcal{D}} & c_{\mathsf{FMT}}, r_0, r_1 \\ \mathsf{sid}, \mathsf{spk}, v_{\mathsf{pub}} & \mathsf{dst}, \mathsf{arlt}, \mathsf{wrlt} & \mathsf{actr}, \mathsf{wctr}, \gamma_{\mathcal{D}} & \end{pmatrix}$.
7. $\pi_{\mathsf{pt}} \leftarrow \mathsf{NIZK.Prove}(\mathsf{pp.crs}_{\mathsf{pt}}, \mathbb{x}, \mathbb{w})$. (See Section 7.2.7 for the NP relation $\mathcal{R}_{\mathsf{pt}}$.)

SEND COMMITMENTS AND PROOF TO RECEIVER. $m_0, \omega_0, \omega_1, r_0, \pi_{\mathsf{pt}}$ ⟶

STEP 5: VERIFY SENDER MESSAGE.
1. Parse m_0 as $(\mathsf{sid}', \mathsf{spk}', v'_{\mathsf{pub}}, \gamma'_{\mathcal{D}}, c_{\mathsf{FMT}}, \mathsf{arlt})$.
2. $\mathbb{x} := (\mathbf{L}.\mathsf{Len}, \mathbf{L}.\mathsf{SNRoot}, \mathbf{L}.\mathsf{CMRoot}, \mathsf{pk}_{\mathsf{SIG}}, \omega_0, \omega_1)$.
3. If $\mathsf{COMM}_{r_0}(m_0) \neq \omega_0$: output $\mathtt{fail}$.
4. If $(\mathsf{sid}', \mathsf{spk}', v'_{\mathsf{pub}}, \gamma'_{\mathcal{D}}) \neq (\mathsf{sid}, \mathsf{spk}, v_{\mathsf{pub}}, \gamma_{\mathcal{D}})$: output $\mathtt{fail}$.
5. If $\mathsf{NIZK.Verify}(\mathsf{pp.crs}_{\mathsf{pt}}, \mathbb{x}, \pi_{\mathsf{pt}}) = 0$: output $\mathtt{fail}$.
6. If $\mathsf{PC.CheckCounter}(\mathcal{P}, \mathsf{arlt}, \mathtt{A}) = 1$: output $\mathtt{fail}$.
7. $\mathcal{P}' \leftarrow \mathsf{PC.Add}(\mathcal{P}, \mathsf{arlt}, \mathtt{A})$, output $\mathcal{P}'$ and continue.
8. $m' \leftarrow \mathsf{FMT.Decrypt}(\mathsf{pp}_{\mathsf{FMT}}, \mathsf{sk}_{\mathsf{FMT}}, c_{\mathsf{FMT}})$.
9. If $m' = \varnothing$: output $\mathtt{null}$ and skip Step 6.

STEP 6: CHECK FOR DOUBLE SPENDING. (only if $m' \neq \varnothing$)
1. Parse m' as $(m_1 \| r_1)$, and m_1 as $(\mathsf{tx}_{\mathsf{p}}, \mathsf{dst}, \mathsf{wrlt})$.
2. Parse tx_{p} as $(\mathsf{ts}, 0, \mathsf{sn}, \mathsf{cm}, v_{\mathsf{pub}}, \mathsf{info}, *)$.
3. If $\mathsf{PC.CheckCounter}(\mathcal{P}, \mathsf{wrlt}, \mathtt{W}) = 1$: output $\mathtt{fail}$.
4. $\mathcal{P}' \leftarrow \mathsf{PC.Add}(\mathcal{P}, \mathsf{wrlt}, \mathtt{W})$, output $\mathcal{P}'$ and continue.
5. $m_{\mathsf{SIG}} := (\mathsf{tx}_{\mathsf{p}}, \mathsf{dst}, \omega_0, \omega_1, r_1, \pi_{\mathsf{pt}})$.
6. $\sigma \leftarrow \mathsf{SIG.Sign}(\mathsf{pp}_{\mathsf{SIG}}, \mathsf{sk}_{\mathsf{SIG}}, m_{\mathsf{SIG}})$.
7. $*' := (*, (\mathsf{pk}_{\mathsf{SIG}}, m_{\mathsf{SIG}}, \sigma))$.
8. $\mathsf{tx}_{\mathsf{pt}} := \mathsf{tx}'_{\mathsf{p}}$ where $\mathsf{tx}'_{\mathsf{p}} := (\mathsf{ts}, 0, \mathsf{sn}, \mathsf{cm}, v_{\mathsf{pub}}, \mathsf{info}, *')$.
9. Search $\mathbf{L}$ for a transaction $\mathsf{tx}'_{\mathsf{pt}}$ s.t. $\mathsf{tx}'_{\mathsf{pt}}.\mathsf{sn} = \mathsf{tx}_{\mathsf{pt}}.\mathsf{sn}$.
10. If such a transaction exists:
    (a) $(\mathsf{tx}_{\mathsf{pun}}, \mathcal{D}') \leftarrow \mathsf{DAM.Punish}(\mathsf{tx}_{\mathsf{pt}}, \mathsf{tx}'_{\mathsf{pt}}, \mathcal{D})$.
    (b) Output $\mathtt{double}$ and $(\mathsf{tx}_{\mathsf{pun}}, \mathcal{D}')$.
11. If no such transaction exists:
    (a) $\mathbf{L}' := (\mathsf{tx}_{\mathsf{p}})$.
    (b) $\mathbf{c} \leftarrow \mathsf{DAP.Receive}^{\mathbf{L}'}(\mathsf{pp.pp}_{\mathsf{DAP}}, (\mathsf{apk}_{\mathbf{c}}, \mathsf{ask}_{\mathbf{c}}))$.
    (c) Output $\mathtt{macro}$ and $(\mathbf{c}, \mathsf{tx}_{\mathsf{pt}})$.

$m', \mathsf{sk}_{\mathsf{FMT}}$ ⟵ SEND COMMITMENT OPENING AND RANDOMNESS TO SENDER.

STEP 7: DEDUCE OUTCOME OF PAYMENT.
1. If $\mathsf{FMT.Decrypt}(\mathsf{pp}_{\mathsf{FMT}}, \mathsf{sk}_{\mathsf{FMT}}, c_{\mathsf{FMT}}) \neq m'$:
   (a) $(\mathbf{t}', \mathsf{tx}_{\mathsf{ref}}) \leftarrow \mathsf{DAM.RefreshTicket}^{\mathbf{L}}(\mathsf{pp}, \mathbf{t}, \mathsf{ask}_{\mathbf{t}}, \mathbf{t}.\mathsf{apk}, v_{\mathsf{pub}}, \mathsf{info})$.
   (b) Output $\mathtt{fail}$ and $(\mathbf{t}', \mathsf{tx}_{\mathsf{ref}})$.
2. If $m' = \varnothing$: output $\mathtt{null}$.
3. If $m' \neq \varnothing$: output $\mathtt{macro}$.

Figure 4: Construction of a DAM scheme (part 2 of 3).

$\mathsf{DAM.VerifyTransaction}^{\mathbf{L}}$

public parameter pp
transaction tx

bit b denoting if tx is valid

If any of the checks below fail, output 0.

1. If tx is a *mint transaction*:
   (a) check that $\mathsf{DAP.VerifyTransaction}^{\mathbf{L}}(\mathsf{pp.pp_{DAP}}, \mathsf{tx_m}) = 1$
2. If tx is a *pour-coin transaction*:
   (a) View tx as a DAP pour transaction $\mathsf{tx_p}$
   (b) Check that $\mathsf{tx_p.info} = (\mathsf{info}', \mathtt{cn})$
   (c) Check that $\mathsf{DAP.VerifyTransaction}^{\mathbf{L}}(\mathsf{pp.pp_{DAP}}, \mathsf{tx_p}) = 1$
3. If tx is a *withdraw transaction*:
   (a) View tx as a DAP pour transaction $\mathsf{tx_p}$.
   (b) Check that $\mathsf{tx_p.info} = (\mathsf{info}', \mathtt{dp})$
   (c) Check that $\mathsf{DAP.VerifyTransaction}^{\mathbf{L}}(\mathsf{pp.pp_{DAP}}, \mathsf{tx_p}) = 1$
4. If tx is a *refresh transaction*:
   (a) View tx as a DAP pour transaction $\mathsf{tx_p}$
   (b) Check that $\mathsf{tx_p.info} = (\mathsf{info}', \mathtt{tk})$
   (c) Check that $\mathsf{DAP.VerifyTransaction}^{\mathbf{L}}(\mathsf{pp.pp_{DAP}}, \mathsf{tx_p}) = 1$
5. If tx is a *pour-ticket transaction*:
   (a) View tx as a DAP pour transaction $\mathsf{tx_p} = (\mathsf{ts}, 0, \mathsf{sn}, \mathsf{cm}, \mathsf{info}, *)$
   (b) Parse $*$ as $(*', (\mathsf{pk_{SIG}}, m_{\mathsf{SIG}}, \sigma))$
   (c) Parse m_{SIG} as $(\mathsf{tx_p}, \mathsf{dst}, \omega_0, \omega_1, r_1, \pi_{\mathsf{pt}})$
   (d) $\mathbb{x} := (\mathbf{L}.\mathsf{Len}, \mathbf{L}.\mathsf{SNRoot}, \mathbf{L}.\mathsf{CMRoot}, \mathsf{pk_{SIG}}, \omega_0, \omega_1)$
   (e) Check that $\mathsf{COMM}_{r_1}(\mathsf{tx_p}, \mathsf{dst}, \mathsf{wrlt}) = \omega_1$
   (f) Check that $\mathsf{SIG.Verify}(\mathsf{pp_{SIG}}, \mathsf{pk_{SIG}}, m_{\mathsf{SIG}}, \sigma) = 1$
   (g) Check that $\mathsf{NIZK.Verify}(\mathsf{pp.crs_{pt}}, \mathbb{x}, \pi_{\mathsf{pt}}) = 1$
   (h) Check that $\mathsf{DAP.VerifyTransaction}^{\mathbf{L}}(\mathsf{pp.pp_{DAP}}, \mathsf{tx_p}) = 1$
6. If tx is a *punish transaction*:
   (a) Parse tx as $(\mathsf{tx_{pt}}, \mathsf{tx}'_{\mathsf{pt}}, \mathsf{sn}, \mathsf{id})$
   (b) Check that $\mathsf{DAM.VerifyTransaction}^{\mathbf{L}}(\mathsf{pp}, \mathsf{tx_{pt}}) = 1$
   (c) Check that $\mathsf{DAM.VerifyTransaction}^{\mathbf{L}}(\mathsf{pp}, \mathsf{tx}'_{\mathsf{pt}}) = 1$
   (d) Check that $(\mathsf{sn}, \mathsf{id}) = \mathsf{DST.GetSecret}(\mathsf{tx_{pt}}, \mathsf{tx}'_{\mathsf{pt}})$

Figure 5: Construction of a DAM scheme (part 3 of 3).

7.4 Security of the construction

The security of our DAM scheme construction relies on the security of the underlying DAP scheme, FMT scheme, and other cryptographic building blocks. Below we state the theorem that asserts this; its proof is provided in Appendix C.

Theorem 7.1 (formal statement of Thm. 1.2)**.** *The DAM scheme specified in Section 7.3 (see Figures 3, 4, 5) is secure, i.e., satisfies the security definition of Definition 6.1 in Section 6.4, provided that the following holds: (1)* DAP *is a DAP scheme secure in the sense of Definition B.2; (2)* FMT *is an FMT scheme secure in the sense of Appendix A.1; (3)* NIZK *is a simulation-extractable non-interactive perfect zero knowledge argument system; (4)* COMM *is a commitment scheme; (5)* CRH *is a collision-resistant hash function; (6)* PRF *is a (collision-resistant) pseudorandom function; (7)* SIG *is a strongly-unforgeable one-time signature scheme.*

A Fractional message transfer

The notion of *fractional message transfer* (FMT) is informally introduced in Section 4; we now formally define it and then give an efficient construction for it. More precisely, we define the notion of an *FMT scheme* (Appendix A.1) and then give an efficient construction for it (Appendix A.2). Afterwards, we define the notion of an *FMT protocol* (Appendix A.3) and then explain how an FMT scheme immediately implies an FMT protocol (Appendix A.4).

Remark A.1 (scheme vs. protocol)**.** Our construction of a DAM scheme uses FMT schemes as a building block (see Section 7). The construction does not directly use an FMT protocol because we interleave the FMT scheme with other building blocks, and because our construction requires the receiver to prove to the sender whether or not he could obtain the encrypted message, which might be restrictive in other uses of the protocol. Still, we find it instructive to also discuss FMT protocols because their security is defined via an ideal functionality, and we show that the security of FMT schemes does imply security relative to that ideal functionality.

Remark A.2 (security definition)**.** Prior work on constructing FMT schemes only required security for random messages, and hence could rely only on the hardness of the CDH problem to prove security [BM89, BR99]. However, we use FMT to encrypt non-random messages, and thus require a security property similar to semantic security. Looking ahead, this is why the security of our construction relies on the DDH assumption and not on the CDH assumption.

Furthermore, in our DAM scheme, the same party can act as both the sender and the receiver. To prove security of our DAM construction, we thus require the FMT scheme to be both *simulatable* and *extractable* in the same experiment; this is similar to the notion of simulation-extractability for non-interactive zero knowledge proof schemes [Sah99, DDO+01].

A.1 Definition of a fractional message transfer scheme

A *fractional message transfer* (FMT) scheme with message space $\mathcal{M}$ and probability space $\mathcal{P} \subseteq [0,1]$ is a tuple of algorithms

$$\mathsf{FMT} = (\mathsf{FMT.Setup}, \mathsf{FMT.Keygen}, \mathsf{FMT.Encrypt}, \mathsf{FMT.Decrypt})$$

with the following syntax.

- PARAMETER SETUP: $\mathsf{FMT.Setup}(1^\lambda) \to \mathsf{pp}_{\mathsf{FMT}}$

  On input a security parameter λ, $\mathsf{FMT.Setup}$ outputs public parameters $\mathsf{pp}_{\mathsf{FMT}}$ for the scheme.

- KEY GENERATION: $\mathsf{FMT.Keygen}(\mathsf{pp}_{\mathsf{FMT}}, p) \to (\mathsf{pk}_{\mathsf{FMT}}, \mathsf{sk}_{\mathsf{FMT}})$

  On input public parameters $\mathsf{pp}_{\mathsf{FMT}}$ and a transfer probability $p \in \mathcal{P}$, $\mathsf{FMT.Keygen}$ outputs a one-time public key $\mathsf{pk}_{\mathsf{FMT}}$ and secret key $\mathsf{sk}_{\mathsf{FMT}}$. (Without loss of generality, both keys contain p in plaintext.)

- MESSAGE ENCRYPTION: $\mathsf{FMT.Encrypt}(\mathsf{pp}_{\mathsf{FMT}}, \mathsf{pk}_{\mathsf{FMT}}, m) \to c$

  On input public parameters $\mathsf{pp}_{\mathsf{FMT}}$, a public key $\mathsf{pk}_{\mathsf{FMT}}$ and a message $m \in \mathcal{M}$, $\mathsf{FMT.Encrypt}$ outputs a ciphertext c.

- MESSAGE DECRYPTION: $\mathsf{FMT.Decrypt}(\mathsf{pp}_{\mathsf{FMT}}, \mathsf{sk}_{\mathsf{FMT}}, c) \to m'$

  On input public parameters $\mathsf{pp}_{\mathsf{FMT}}$, a secret key $\mathsf{sk}_{\mathsf{FMT}}$ and a ciphertext c, $\mathsf{FMT.Decrypt}$ outputs a message m' that equals m or $\varnothing$.

An FMT scheme satisfies the correctness and security properties described below.

Correctness. An FMT scheme is *correct* if for every security parameter λ, public parameters $\mathsf{pp}_{\mathsf{FMT}} \in \mathsf{FMT.Setup}(1^\lambda)$, transfer probability $p \in \mathcal{P}$, key pair $(\mathsf{pk}_{\mathsf{FMT}}, \mathsf{sk}_{\mathsf{FMT}}) \in \mathsf{FMT.Keygen}(\mathsf{pp}_{\mathsf{FMT}}, p)$, and message $m \in \mathcal{M}$,

$$\mathsf{FMT.Decrypt}(\mathsf{pp}_{\mathsf{FMT}}, \mathsf{sk}_{\mathsf{FMT}}, \mathsf{FMT.Encrypt}(\mathsf{pp}_{\mathsf{FMT}}, \mathsf{pk}_{\mathsf{FMT}}, m)) = \begin{cases} m & \text{w.p.} \quad p \\ \varnothing & \text{w.p.} \quad 1-p \end{cases}$$

where the probability is taken over the randomness of $\mathsf{FMT.Encrypt}$ (and $\mathsf{FMT.Decrypt}$ is deterministic).

Security. An FMT scheme is *secure* if there exist algorithms $(\mathsf{FMT.SimSetup}, \mathsf{FMT.SimKeygen}, \mathsf{FMT.SimEncrypt}, \mathsf{FMT.ExtDecrypt}, \mathsf{FMT.SimDecrypt})$ that yield the properties of *fractional hiding* and *fractional binding*, as described below. (Informally, these correspond to *sender security* and *receiver security* of an FMT protocol, respectively; see Appendix A.3.)

- SIMULATION OF PARAMETER SETUP: $\mathsf{FMT.SimSetup}(1^\lambda) \to (\mathsf{pp}_{\mathsf{FMT}}, \mathsf{td}_{\mathsf{FMT}})$

  On input a security parameter λ, $\mathsf{FMT.SimSetup}$ outputs simulated public parameters $\mathsf{pp}_{\mathsf{FMT}}$ and a simulation trapdoor $\mathsf{td}_{\mathsf{FMT}}$.

- SIMULATION OF KEY GENERATION: $\mathsf{FMT.SimKeygen}(\mathsf{pp}_{\mathsf{FMT}}, \mathsf{td}_{\mathsf{FMT}}, p) \to (\mathsf{pk}_{\mathsf{FMT}}, \mathsf{sk}_{\mathsf{FMT}})$

  On input simulated public parameters $\mathsf{pp}_{\mathsf{FMT}}$, a simulation trapdoor $\mathsf{td}_{\mathsf{FMT}}$, and a transfer probability $p \in \mathcal{P}$, $\mathsf{FMT.SimKeygen}$ outputs a simulated key pair $(\mathsf{pk}_{\mathsf{FMT}}, \mathsf{sk}_{\mathsf{FMT}})$.

- SIMULATION OF MESSAGE ENCRYPTION: $\mathsf{FMT.SimEncrypt}(\mathsf{pp}_{\mathsf{FMT}}, \mathsf{td}_{\mathsf{FMT}}, \mathsf{pk}_{\mathsf{FMT}}, b, m') \to c$

  On input simulated public parameters $\mathsf{pp}_{\mathsf{FMT}}$, a simulation trapdoor $\mathsf{td}_{\mathsf{FMT}}$, a public key $\mathsf{pk}_{\mathsf{FMT}}$, a bit b, and a message $m' \in \mathcal{M}$, $\mathsf{FMT.SimEncrypt}$ outputs a ciphertext c.

- EXTRACTION OF MESSAGE FOR DECRYPTION: $\mathsf{FMT.ExtDecrypt}(\mathsf{pp}_{\mathsf{FMT}}, \mathsf{td}_{\mathsf{FMT}}, \mathsf{sk}_{\mathsf{FMT}}, c) \to (m, \mathsf{sk}'_{\mathsf{FMT}})$

  On input sumulated public parameters $\mathsf{pp}_{\mathsf{FMT}}$, a simulation trapdoor $\mathsf{td}_{\mathsf{FMT}}$, a secret key $\mathsf{sk}_{\mathsf{FMT}}$, and a ciphertext c, $\mathsf{FMT.ExtDecrypt}$ outputs a message m such that $m = \mathsf{FMT.Decrypt}(\mathsf{pp}_{\mathsf{FMT}}, \mathsf{sk}_{\mathsf{FMT}}, c)$.

- SIMULATION FOR DECRYPTION VERIFICATION: $\mathsf{FMT.SimDecrypt}(\mathsf{pp}_{\mathsf{FMT}}, \mathsf{td}_{\mathsf{FMT}}, \mathsf{sk}_{\mathsf{FMT}}, b) \to \mathsf{sk}'_{\mathsf{FMT}}$

  On input public parameters $\mathsf{pp}_{\mathsf{FMT}}$, a simulation trapdoor $\mathsf{td}_{\mathsf{FMT}}$, a secret key $\mathsf{sk}_{\mathsf{FMT}}$, and a bit b, $\mathsf{FMT.SimDecrypt}$ outputs a simulated secret key $\mathsf{sk}'_{\mathsf{FMT}}$.

We now use the above algorithms to specify the two security properties.

- **Fractional hiding.** Informally, fractional hiding says that an honest encryptor transferring a message m can be sure that the decryptor, who knows the secret key, learns m with probability exactly p (and $\varnothing$ with probability $1-p$), even if the public key was maliciously generated. In detail, given an efficient adversary $\mathcal{A}_{\mathsf{FH}}$ and transfer probability $p \in \mathcal{P}$, we define the following two experiments.

  - $\mathtt{Real}(\mathcal{A}_{\mathsf{FH}}, p)$
    1. Sample $\mathsf{pp}_{\mathsf{FMT}} \leftarrow \mathsf{FMT.Setup}(1^\lambda)$ and give $\mathsf{pp}_{\mathsf{FMT}}$ to $\mathcal{A}_{\mathsf{FH}}$.
    2. The adversary $\mathcal{A}_{\mathsf{FH}}$ replies with $(\mathsf{pk}_{\mathsf{FMT}}, m)$. (If $\mathsf{pk}_{\mathsf{FMT}}$ is not a valid public key or $m \notin \mathcal{M}$, abort.)
    3. Sample $c \leftarrow \mathsf{FMT.Encrypt}(\mathsf{pp}_{\mathsf{FMT}}, \mathsf{pk}_{\mathsf{FMT}}, m)$ and give c to $\mathcal{A}_{\mathsf{FH}}$.
  - $\mathtt{Ideal}(\mathcal{A}_{\mathsf{FH}}, p)$
    1. Sample $(\mathsf{pp}_{\mathsf{FMT}}, \mathsf{td}_{\mathsf{FMT}}) \leftarrow \mathsf{FMT.SimSetup}(1^\lambda)$ and give $\mathsf{pp}_{\mathsf{FMT}}$ to $\mathcal{A}_{\mathsf{FH}}$.
    2. The adversary $\mathcal{A}_{\mathsf{FH}}$ replies with $(\mathsf{pk}_{\mathsf{FMT}}, m)$. (If $\mathsf{pk}_{\mathsf{FMT}}$ is not a valid public key or $m \notin \mathcal{M}$, abort.)
    3. Sample $b \in \{0,1\}$ such that $b = 1$ with probability p.
    4. If $b = 1$, set $m' := m$; if $b = 0$, sample a message m' in $\mathcal{M}$ at random.
    5. Sample $c \leftarrow \mathsf{FMT.SimEncrypt}(\mathsf{pp}_{\mathsf{FMT}}, \mathsf{td}_{\mathsf{FMT}}, \mathsf{pk}_{\mathsf{FMT}}, b, m')$ and give c to $\mathcal{A}_{\mathsf{FH}}$.

  Denote by $\mathsf{Output}\,(\mathtt{Real}(\mathcal{A}_{\mathsf{FH}}))$ and $\mathsf{Output}\,(\mathtt{Ideal}(\mathcal{A}_{\mathsf{FH}}))$ the output of $\mathcal{A}_{\mathsf{FH}}$ in the first and second experiments, respectively. An FMT scheme is *fractional hiding* if, for every stateful efficient adversary $\mathcal{A}_{\mathsf{FH}}$ and transfer probability $p \in \mathcal{P}$, $\mathsf{Output}\,(\mathtt{Real}(\mathcal{A}_{\mathsf{FH}}))$ and $\mathsf{Output}\,(\mathtt{Ideal}(\mathcal{A}_{\mathsf{FH}}))$ are computationally indistinguishable.

- **Fractional binding.** Informally, fractional binding says that, for every $p' \neq p$, a malicious encryptor cannot produce a valid ciphertext that decrypts with probability p' to a valid message (i.e., not $\varnothing$). In detail, given an efficient adversary $\mathcal{A}_{\mathsf{FB}}$ and transfer probability $p \in \mathcal{P}$, we define the following two experiments.

  - $\mathtt{Real}(\mathcal{A}_{\mathsf{FB}}, p)$
    1. Sample $\mathsf{pp}_{\mathsf{FMT}} \leftarrow \mathsf{FMT.Setup}(1^\lambda)$ and give $\mathsf{pp}_{\mathsf{FMT}}$ to $\mathcal{A}_{\mathsf{FB}}$.

2. Sample $(\mathsf{pk}_{\mathsf{FMT}}, \mathsf{sk}_{\mathsf{FMT}}) \leftarrow \mathsf{FMT.Keygen}(\mathsf{pp}_{\mathsf{FMT}}, p)$ and give $\mathsf{pk}_{\mathsf{FMT}}$ to $\mathcal{A}_{\mathsf{FB}}$.
3. The adversary $\mathcal{A}_{\mathsf{FB}}$ replies with a ciphertext c. (If c is an invalid ciphertext, abort.)
4. Sample $m \leftarrow \mathsf{FMT.Decrypt}(\mathsf{pp}_{\mathsf{FMT}}, \mathsf{sk}_{\mathsf{FMT}}, c)$.
5. Output $(m, \mathsf{sk}_{\mathsf{FMT}})$.

– $\mathtt{Ideal}(\mathcal{A}_{\mathsf{FB}}, p)$
1. Sample $(\mathsf{pp}_{\mathsf{FMT}}, \mathsf{td}_{\mathsf{FMT}}) \leftarrow \mathsf{FMT.SimSetup}(1^\lambda)$ and give $\mathsf{pp}_{\mathsf{FMT}}$ to $\mathcal{A}_{\mathsf{FB}}$.
2. Sample $(\mathsf{pk}_{\mathsf{FMT}}, \mathsf{sk}_{\mathsf{FMT}}) \leftarrow \mathsf{FMT.SimKeygen}(\mathsf{pp}_{\mathsf{FMT}}, \mathsf{td}_{\mathsf{FMT}}, p)$ and give $\mathsf{pk}_{\mathsf{FMT}}$ to $\mathcal{A}_{\mathsf{FB}}$.
3. The adversary $\mathcal{A}_{\mathsf{FB}}$ replies with a ciphertext c. (If c is an invalid ciphertext, abort.)
4. Sample $m \leftarrow \mathsf{FMT.ExtDecrypt}(\mathsf{pp}_{\mathsf{FMT}}, \mathsf{td}_{\mathsf{FMT}}, \mathsf{sk}_{\mathsf{FMT}}, c)$.
5. Sample a bit b such that $b = 1$ with probability p and 0 otherwise.
6. Sample $\mathsf{sk}'_{\mathsf{FMT}} \leftarrow \mathsf{FMT.SimDecrypt}(\mathsf{pp}_{\mathsf{FMT}}, \mathsf{td}_{\mathsf{FMT}}, \mathsf{sk}_{\mathsf{FMT}}, b)$
7. Output $(m, \mathsf{sk}'_{\mathsf{FMT}})$ if $b = 1$, and $(\varnothing, \mathsf{sk}'_{\mathsf{FMT}})$ otherwise.

Denote by $\mathsf{Output}\,(\mathtt{Real}(\mathcal{A}_{\mathsf{FB}}))$ and $\mathsf{Output}\,(\mathtt{Ideal}(\mathcal{A}_{\mathsf{FB}}))$ the output of the first and second experiments, respectively. An FMT scheme is *fractional binding* if, for every efficient adversary $\mathcal{A}_{\mathsf{FB}}$ and transfer probability $p \in \mathcal{P}$, $\mathsf{Output}\,(\mathtt{Real}(\mathcal{A}_{\mathsf{FB}}))$ and $\mathsf{Output}\,(\mathtt{Ideal}(\mathcal{A}_{\mathsf{FB}}))$ are computationally indistinguishable.

A.2 Construction of a fractional message transfer scheme

We give an efficient construction of an FMT scheme, for the case when the transfer probability p equals $1/n$ for some integer n with $1 \le n < q$ and q a cryptographically-large prime. (This suffices for the purposes of this paper.) Our construction uses the following two ingredients.

- A group generator $\mathsf{GroupGen}$ that, on input a security parameter λ (represented in unary), outputs a tuple $(\mathbb{G}, q, g)$ that describes a group $\mathbb{G}$ of prime order q generated by g. We assume that the DDH problem is hard for $\mathsf{GroupGen}$, that is, the following two distribution families are computationally indistinguishable:

$$\left\{(\mathbb{G}, q, g, g^a, g^b, g^c) \;\middle|\; \begin{array}{r}(\mathbb{G}, q, g) \leftarrow \mathsf{GroupGen}(1^\lambda) \\ a, b, c \leftarrow \mathbb{Z}_q\end{array}\right\}_{\lambda \in \mathbb{N}} \text{and} \left\{(\mathbb{G}, q, g, g^a, g^b, g^{ab}) \;\middle|\; \begin{array}{r}(\mathbb{G}, q, g) \leftarrow \mathsf{GroupGen}(1^\lambda) \\ a, b \leftarrow \mathbb{Z}_q\end{array}\right\}_{\lambda \in \mathbb{N}}.$$

- A non-interactive zero-knowledge proof of knowledge system for (specific) NP relations. We denote it as $\mathsf{NIZK} = (\mathsf{NIZK.Setup}, \mathsf{NIZK.Prove}, \mathsf{NIZK.Verify})$ and its syntax is as follows: given a security parameter λ and the specification of an NP relation $\mathcal{R}$, $\mathsf{NIZK.Setup}$ outputs a common reference string crs; given crs and an instance-witness pair $(\mathbb{x}, \mathbb{w}) \in \mathcal{R}$, $\mathsf{NIZK.Prove}$ outputs a proof π; and given crs, instance $\mathbb{x}$, and proof π, $\mathsf{NIZK.Verify}$ outputs a decision bit b. We use such a proof system only for these two simple NP relations over elements in $\mathbb{G}$:
  - $\mathcal{R}_{\mathsf{K}} := \left\{\big((z, g_1, g_2), (s, \alpha)\big) \text{ s.t. } z = {g_1}^{-s} {g_2}^{\alpha}\right\}$, and
  - $\mathcal{R}_{\mathsf{E}} := \left\{\big((z_1, z_2, g_1, g_2), (m, r)\big) \text{ s.t. } z_1 = {g_1}^{r} \text{ and } z_2 = m \cdot {g_2}^{r}\right\}$.

  For these relations, we do not need generic proof systems; instead, efficient proof systems tailored to proving statements about discrete logarithms suffice, as we now explain.
  - *For $\mathcal{R}_{\mathsf{K}}$:* In a prime-order group every non-identity element is a generator; so $\mathcal{R}_{\mathsf{K}}$ is the relation of Pedersen commitment and decommitment pairs [Ped91]. One can obtain an efficient proof system for this by applying the Fiat–Shamir transform to known Σ-protocols [Oka92]. For the resulting proof system: (a) proving requires two exponentiations, (b) verification requires three exponentiations, and (c) the proof consists of three scalars in $\mathbb{Z}_q$.
  - *For $\mathcal{R}_{\mathsf{E}}$:* The relation $\mathcal{R}_{\mathsf{E}}$ consists of instances that are Elgamal ciphertexts and witnesses that are the corresponding underlying message m and randomness r. One can verify that, for Elgamal ciphertexts, fixing the choice of randomness r fixes the message m [SJ00]. Hence one only needs to prove knowledge of r to also prove knowledge of m. One can obtain an efficient proof system to do this by applying the Fiat–Shamir transform to known Σ-protocols [CEv87, Sch91]. For the resulting proof system: (a) proving requires one exponentiation, (b) verification requires two exponentiations, and (c) the proof consists of two scalars in $\mathbb{Z}_q$.

To satisfy the notion of "simulation-extractability" for FMT schemes mentioned in Remark A.2, we require that the NIZKs above be simulation-extractable. Since these NIZKs are instantiated via the Fiat–Shamir transform over Σ-protocols, we obtain simulation-extractability for free.

We now describe our construction of an FMT scheme in terms of the above ingredients, and then state our theorem.

$\underline{\mathsf{FMT.Setup}(1^\lambda)}$
1. Sample a group description $(\mathbb{G}, q, g) \leftarrow \mathsf{GroupGen}(1^\lambda)$.
2. Sample g_0 at random in $\mathbb{G}$.
3. Compute $\mathsf{crs_K} \leftarrow \mathsf{NIZK.Setup}(1^\lambda, \mathcal{R}_\mathsf{K})$.
4. Compute $\mathsf{crs_E} \leftarrow \mathsf{NIZK.Setup}(1^\lambda, \mathcal{R}_\mathsf{E})$.
5. Output public parameters $\mathsf{pp_{FMT}} := (\mathbb{G}, q, g, g_0, \mathsf{crs_K}, \mathsf{crs_E})$.

$\underline{\mathsf{FMT.Keygen}(\mathsf{pp_{FMT}}, 1/n)}$
1. Parse $\mathsf{pp_{FMT}}$ as $(\mathbb{G}, q, g, g_0, \mathsf{crs_K}, \mathsf{crs_E})$.
2. Sample s uniformly in $\{1, \ldots, n\}$.
3. Sample α uniformly in $\mathbb{Z}_q$.
4. Set $h := {g_0}^{-s} g^\alpha$.
5. Set $\pi_\mathsf{K} := \mathsf{NIZK.Prove}(\mathsf{crs_K}, (h, g_0, g), (s, \alpha))$.
6. Set $\mathsf{pk_{FMT}} := (\mathsf{pp_{FMT}}, 1/n, h, \pi_\mathsf{K})$.
7. Set $\mathsf{sk_{FMT}} := (\mathsf{pp_{FMT}}, 1/n, h, \pi_\mathsf{K}, s, \alpha)$.
8. Output key pair $(\mathsf{pk_{FMT}}, \mathsf{sk_{FMT}})$.

$\underline{\mathsf{FMT.Encrypt}(\mathsf{pp_{FMT}}, \mathsf{pk_{FMT}}, m)}$
1. Parse $\mathsf{pp_{FMT}}$ as $(\mathbb{G}, q, g, g_0, \mathsf{crs_K}, \mathsf{crs_E})$.
2. Parse $\mathsf{pk_{FMT}}$ as a tuple $(1/n, h, \pi_\mathsf{K})$.
3. If $\mathsf{NIZK.Verify}(\mathsf{crs_K}, (h, g_0, g), \pi_\mathsf{K}) = 0$: halt and output $\perp$.
4. Sample t at random in $\{1, \ldots, n\}$.
5. Sample r at random in $\mathbb{Z}_q$.
6. Set $c_1 := g^r$ and $c_2 := m \cdot (g_0^t h)^r$.
7. Set $\pi_\mathsf{E} := \mathsf{NIZK.Prove}(\mathsf{crs_E}, (c_1, c_2, g, {g_0}^t h), (m, r))$.
8. Output ciphertext $c := (t, c_1, c_2, \pi_\mathsf{E})$.

$\underline{\mathsf{FMT.Decrypt}(\mathsf{pp_{FMT}}, \mathsf{sk_{FMT}}, c)}$
1. Parse $\mathsf{pp_{FMT}}$ as $(\mathbb{G}, q, g, g_0, \mathsf{crs_K}, \mathsf{crs_E})$.
2. Parse $\mathsf{sk_{FMT}}$ as $(1/n, h, \pi_\mathsf{K}, s, \alpha)$.
3. Parse c as $(t, c_1, c_2, \pi_\mathsf{E})$.
4. If $t \notin \{1, \ldots, n\}$: halt and output $\perp$.
5. If $\mathsf{NIZK.Verify}(\mathsf{crs_E}, (c_1, c_2, g, {g_0}^t h), \pi_\mathsf{E}) = 0$: halt and output $\perp$.
6. If $t \neq s$: set $m' := \varnothing$, and output m'.
7. If $t = s$: set $m' := c_2 / {c_1}^\alpha$, and output m'.

Theorem A.3 (formal statement of Thm. 1.3)**.** *The above construction is a correct and secure FMT scheme when assuming (1) the hardness of* DDH *for* GroupGen*, and (2) the security of* NIZK *for the* NP *relations $\mathcal{R}_\mathsf{K}$ and $\mathcal{R}_\mathsf{E}$. The message space $\mathcal{M}$ of the FMT scheme is the group $\mathbb{G}$ sampled by* GroupGen*, and the probability space $\mathcal{P}$ is the set of rationals $1/n$ such that $1 \leq n < |\mathbb{G}|$.*

Remark A.4. In the random oracle model, we can efficiently instantiate the NIZKs used in Theorem A.3 by applying the Fiat–Shamir transform to the aforementioned Σ-protocols. The resulting FMT scheme has public keys consisting of one group element in $\mathbb{G}$ and three scalars in $\mathbb{Z}_q$, and ciphertexts of two group elements in $\mathbb{G}$ and two scalars in $\mathbb{Z}_q$.

To prove the theorem, we first argue correctness and then argue security.

Correctness. An honestly-generated ciphertext c is a tuple $(t, c_1, c_2, \pi_\mathsf{E})$ where t is random in $\{1, \ldots, n\}$, $c_1 = g^r$, $c_2 = m \cdot (g_0^t h)^r$, and π_E is a valid proof. The decryption algorithm gets as input a private key $\mathsf{sk_{FMT}}$, which contains (s, α) such that $h = {g_0}^{-s} g^\alpha$; after verifying that $t \in \{1, \ldots, n\}$ and π_E is valid, the decryption algorithm first checks if $t = s$. If so, he computes

$$c_2 / c_1^\alpha = m \cdot (g_0^{r(t-s)} g^{r\alpha}) / g^{r\alpha} = m \cdot (g^{r\alpha}) / g^{r\alpha} = m.$$

If not, he outputs $\varnothing$. Since t and s are random in $\{1, \ldots, n\}$, $t = s$ with probability $1/n$, and the receiver obtains m and $\varnothing$ with the claimed probabilities, as required.

Security. The NIZK scheme is a zero knowledge proof of knowledge, and so admits the following algorithms that enable witness extraction or proof simulation, depending on the "mode" of the simulated common reference string.

- $\mathsf{NIZK.SimSetup}(1^\lambda, \mathcal{R}) \to (\mathsf{crs}, \mathsf{td_{NIZK}})$. On input a security parameter λ and the specification of an NP relation $\mathcal{R}$, NIZK.SimSetup outputs a simulated common reference string crs and the corresponding trapdoor $\mathsf{td_{NIZK}}$.
- $\mathsf{NIZK.Extract}(\mathsf{crs}, \mathsf{td_{NIZK}}, \mathbb{x}, \pi) \to \mathbb{w}$. On input a simulated common reference string crs, a simulation trapdoor $\mathsf{td_{NIZK}}$, an NP instance $\mathbb{x}$ and a proof π, NIZK.Extract outputs the associated witness $\mathbb{w}$.
- $\mathsf{NIZK.Simulate}(\mathsf{crs}, \mathsf{td_{NIZK}}, \mathbb{x}) \to \pi$. On input a simulated common reference string crs, a simulation trapdoor $\mathsf{td_{NIZK}}$, and an NP instance $\mathbb{x}$, NIZK.Simulate outputs a simulated proof π.

We now construct the simulation and extraction algorithms (FMT.SimSetup, FMT.SimKeygen, FMT.SimEncrypt, FMT.ExtDecrypt, FMT.SimDecrypt) required to show fractional hiding and fractional binding.

$\underline{\mathsf{FMT.SimSetup}(1^\lambda)}$

1. Sample a group description $(\mathbb{G}, q, g) \leftarrow \mathsf{GroupGen}(1^\lambda)$.
2. Sample x at random in $\mathbb{Z}_q$, and set $g_0 = g^x$.
3. Compute $(\mathsf{crs_K}, \mathsf{td_K}) \leftarrow \mathsf{NIZK.SimSetup}(1^\lambda, \mathcal{R}_\mathsf{K})$.
4. Compute $(\mathsf{crs_E}, \mathsf{td_E}) \leftarrow \mathsf{NIZK.SimSetup}(1^\lambda, \mathcal{R}_\mathsf{E})$.
5. Set $\mathsf{td_{FMT}} := (\mathsf{td_K}, \mathsf{td_E}, x)$.
6. Set $\mathsf{pp_{FMT}} := (\mathbb{G}, q, g, g_0, \mathsf{crs_K}, \mathsf{crs_E})$.
7. Output $(\mathsf{pp_{FMT}}, \mathsf{td_{FMT}})$.

$\underline{\mathsf{FMT.SimKeygen}(\mathsf{pp_{FMT}}, \mathsf{td_{FMT}}, 1/n)}$

1. Parse $\mathsf{pp_{FMT}}$ as $(\mathbb{G}, q, g, g_0, \mathsf{crs_K}, \mathsf{crs_E})$.
2. Parse $\mathsf{td_{FMT}}$ as $(\mathsf{td_K}, \mathsf{td_E}, x)$.
3. Sample s uniformly in $\{1, \ldots, n\}$.
4. Sample α uniformly in $\mathbb{Z}_q$.
5. Set $h := {g_0}^{-s} g^\alpha$.
6. Set $\pi_\mathsf{K} := \mathsf{NIZK.Simulate}(\mathsf{crs_K}, \mathsf{td_K}, (h, g_0, g))$.
7. Set $\mathsf{pk_{FMT}} := (\mathsf{pp_{FMT}}, 1/n, h, \pi_\mathsf{K})$.
8. Set $\mathsf{sk_{FMT}} := (\mathsf{pp_{FMT}}, 1/n, h, \pi_\mathsf{K}, s, \alpha)$.
9. Output key pair $(\mathsf{pk_{FMT}}, \mathsf{sk_{FMT}})$.

$\underline{\mathsf{FMT.ExtDecrypt}(\mathsf{pp_{FMT}}, \mathsf{td_{FMT}}, \mathsf{sk_{FMT}}, c)}$

1. Parse $\mathsf{pp_{FMT}}$ as $(\mathbb{G}, q, g, g_0, \mathsf{crs_K}, \mathsf{crs_E})$.
2. Parse $\mathsf{td_{FMT}}$ as $(\mathsf{td_K}, \mathsf{td_E}, x)$.
3. Parse $\mathsf{sk_{FMT}}$ as $(1/n, h, \pi_\mathsf{K}, s, \alpha)$.
4. Compute $(m, r) \leftarrow \mathsf{NIZK.Extract}(\mathsf{crs_E}, \mathsf{td_E}, (c_1, c_2, g, {g_0}^t h), \pi_\mathsf{E})$.
5. Output m.

$\underline{\mathsf{FMT.SimEncrypt}(\mathsf{pp_{FMT}}, \mathsf{td_{FMT}}, \mathsf{pk_{FMT}}, b, m')}$

1. Parse $\mathsf{pp_{FMT}}$ as $(\mathbb{G}, q, g, g_0, \mathsf{crs_K}, \mathsf{crs_E})$.
2. Parse $\mathsf{td_{FMT}}$ as $(\mathsf{td_K}, \mathsf{td_E}, x)$.
3. Parse $\mathsf{pk_{FMT}}$ as $(1/n, h, \pi_\mathsf{K})$.
4. Compute $(s, \alpha) \leftarrow \mathsf{NIZK.Extract}(\mathsf{crs_K}, \mathsf{td_K}, (h, g_0, g), \pi_\mathsf{K})$.
5. If $s \notin \{1, \ldots, n\}$:
   - sample t at random in $\{1, \ldots, n\}$;
   - sample m'' at random in $\mathcal{M}$.
6. If $s \in \{1, \ldots, n\}$:
   - if $b = 1$, then set $t := s$;
   - if $b = 0$, then set t to any value in $\{1, \ldots, n\} \setminus \{s\}$;
   - set $m'' := m'$.
7. Sample r at random in $\mathbb{Z}_q$.
8. Set $c_1 := g^r$ and $c_2 := m'' \cdot (g_0^t h)^r$.
9. Set $\pi_\mathsf{E} := \mathsf{NIZK.Simulate}(\mathsf{crs_E}, \mathsf{td_E}, (c_1, c_2, g, {g_0}^t h))$.
10. Output ciphertext $c := (t, c_1, c_2, \pi_\mathsf{E})$.

$\underline{\mathsf{FMT.SimDecrypt}(\mathsf{pp_{FMT}}, \mathsf{td_{FMT}}, \mathsf{sk_{FMT}}, b)}$

1. Parse $\mathsf{pp_{FMT}}$ as $(\mathbb{G}, q, g, g_0, \mathsf{crs_K}, \mathsf{crs_E})$.
2. Parse $\mathsf{td_{FMT}}$ as $(\mathsf{td_K}, \mathsf{td_E}, x)$.
3. Parse $\mathsf{sk_{FMT}}$ as $(1/n, h, \pi_\mathsf{K}, s, \alpha)$.
4. If $b = 0$:
   - sample s' such that $s \neq s'$, and set $\alpha' := (\alpha - sx) + s'x$.
   - set $\mathsf{sk'_{FMT}} := (1/n, h, \pi_\mathsf{K}, s', \alpha')$.
   - output $\mathsf{sk'_{FMT}}$.
5. If $b = 1$: output $\mathsf{sk_{FMT}}$.

Before arguing fractional hiding and fractional binding, we state and sketch the proof of two claims.

Claim A.5. *The following two distribution ensembles are computationally indistinguishable:*

$$\left\{\mathsf{pp_{FMT}} \;\middle|\; \mathsf{pp_{FMT}} \leftarrow \mathsf{FMT.Setup}(1^\lambda)\right\}_{\lambda\in\mathbb{N}} \text{ and}$$
$$\left\{\mathsf{pp_{FMT}} \;\middle|\; (\mathsf{pp_{FMT}}, \mathsf{td_{FMT}}) \leftarrow \mathsf{FMT.SimSetup}(1^\lambda)\right\}_{\lambda\in\mathbb{N}} .$$

Proof sketch. The two sets of public parameters differ only in the NIZK common reference strings. In the former case, these are sampled normally, while in the latter case they are simulated. By the security of the NIZK scheme, these common reference strings are computationally indistinguishable. □

Claim A.6. *For every transfer probability $p \in \mathcal{P}$, the following two distribution ensembles are computationally indistinguishable:*

$$\left\{(\mathsf{pp_{FMT}}, \mathsf{pk_{FMT}}) \;\middle|\; \begin{array}{r} \mathsf{pp_{FMT}} \leftarrow \mathsf{FMT.Setup}(1^\lambda) \\ (\mathsf{pk_{FMT}}, \mathsf{sk_{FMT}}) \leftarrow \mathsf{FMT.Keygen}(\mathsf{pp_{FMT}}, p) \end{array}\right\}_{\lambda\in\mathbb{N}} \text{ and}$$
$$\left\{(\mathsf{pp_{FMT}}, \mathsf{pk_{FMT}}) \;\middle|\; \begin{array}{r} (\mathsf{pp_{FMT}}, \mathsf{td_{FMT}}) \leftarrow \mathsf{FMT.SimSetup}(1^\lambda) \\ (\mathsf{pk_{FMT}}, \mathsf{sk_{FMT}}) \leftarrow \mathsf{FMT.SimKeygen}(\mathsf{pp_{FMT}}, \mathsf{td_{FMT}}, p) \end{array}\right\}_{\lambda\in\mathbb{N}} .$$

Proof sketch. The difference between the two is that the public key output by FMT.SimKeygen contains a simulated proof instead of an actual proof. By the security of the NIZK scheme, these are computationally indistinguishable. □

Fractional hiding. Fractional hiding of our construction is implied by the following claim.

Claim A.7. *For every stateful efficient adversary $\mathcal{A}_\mathsf{FH}$ and transfer probability $p \in \mathcal{P}$, the following distribution ensembles are indistinguishable:*

$$\left\{\mathsf{out} \;\middle|\; \begin{array}{r} \mathsf{pp_{FMT}} \leftarrow \mathsf{FMT.Setup}(1^\lambda) \\ (\mathsf{pk_{FMT}}, m) \leftarrow \mathcal{A}_\mathsf{FH}(\mathsf{pp_{FMT}}) \\ c \leftarrow \mathsf{FMT.Encrypt}(\mathsf{pp_{FMT}}, \mathsf{pk_{FMT}}, m) \\ \mathsf{out} \leftarrow \mathcal{A}_\mathsf{FH}(c) \end{array}\right\}_{\lambda\in\mathbb{N}} \text{ and}$$

$$\left\{ \mathsf{out} \;\middle|\; \begin{array}{r} (\mathsf{pp}_{\mathsf{FMT}}, \mathsf{td}_{\mathsf{FMT}}) \leftarrow \mathsf{FMT.SimSetup}(1^\lambda) \\ (\mathsf{pk}_{\mathsf{FMT}}, m) \leftarrow \mathcal{A}_{\mathsf{FH}}(\mathsf{pp}_{\mathsf{FMT}}) \\ b \leftarrow \text{“1 } \textit{w.p. } p \textit{ and } 0 \textit{ otherwise”} \\ \textit{If } b = 1\textit{, set } m' := m\textit{; else sample } m' \textit{ uniformly in } \mathcal{M} \\ c \leftarrow \mathsf{FMT.SimEncrypt}(\mathsf{pp}_{\mathsf{FMT}}, \mathsf{td}_{\mathsf{FMT}}, \mathsf{pk}_{\mathsf{FMT}}, b, m') \\ \mathsf{out} \leftarrow \mathcal{A}_{\mathsf{FH}}(c) \end{array} \right\}_{\lambda \in \mathbb{N}} .$$

In both experiments above, if $\mathsf{pk}_{\mathsf{FMT}}$ *is not a valid public key or* $m \notin \mathcal{M}$ *(both produced by* $\mathcal{A}_{\mathsf{FH}}$*), then abort.*

Proof sketch. For ease of exposition, in this proof we "wrap" the sampling of b, assignment of m', and computation of c by FMT.SimEncrypt into a single procedure, which we call SimExperiment.

By Claim A.5, the public parameters used in either experiment are computationally indistinguishable. Hence, the public keys $\mathsf{pk}_{\mathsf{FMT}}$ generated by the adversary $\mathcal{A}_{\mathsf{FH}}$ in either case must be computationally indistinguishable. Now we are left to argue the indistinguishability of the outputs of FMT.Encrypt and SimExperiment. Consider the following sequence of hybrids on this part of the experiment, where we change how t is sampled:

- $\mathcal{H}_0$: FMT.Encrypt. (That is, sample t at random in $\{1, \ldots, n\}$.)
- $\mathcal{H}_1$: Sample b as in SimExperiment, but set $m' := m$ regardless of b. Then modify the algorithm FMT.Encrypt into an algorithm FMT.Encrypt$'$ that additionally takes as input b and the trapdoor $\mathsf{td}_{\mathsf{FMT}}$, and proceeds as follows.
  1. Parse $\mathsf{td}_{\mathsf{FMT}}$ as $(\mathsf{td}_{\mathsf{K}}, \mathsf{td}_{\mathsf{E}})$.
  2. Compute $(s, \alpha) \leftarrow \mathsf{NIZK.Extract}(\mathsf{crs}_{\mathsf{K}}, \mathsf{td}_{\mathsf{K}}, (h, g_0, g), \pi_{\mathsf{K}})$.
  3. If $s \notin \{1, \ldots, n\}$, then sample t at random in $\{1, \ldots, n\}$ (as before).
  4. If $s \in \{1, \ldots, n\}$ and $b = 1$, then set $t := s$.
  5. If $s \in \{1, \ldots, n\}$ and $b = 0$, then sample t at random in $\{1, \ldots, n\} \setminus \{s\}$.

  The last two steps imply that FMT.Encrypt$'$ does not merely sample t at random in $\{1, \ldots, n\}$; rather, it sets t depending on s. Recall that, in contrast, FMT.Encrypt does sample t at random (see its Step 4). However, this does not change the distribution of t in the two experiments; in both, t is distributed uniformly in $\{1, \ldots, n\}$.
- $\mathcal{H}_2$: Make the following changes to the previous hybrid.
  - Instead of setting $m' := m$ regardless of b, assign m' as in SimExperiment (depending on b), and then pass this as input to FMT.Encrypt$''$, which is a further modification of FMT.Encrypt$'$, as we now describe.
  - Modify Step 3 of FMT.Encrypt$'$ as follows:
    If $s \notin \{1, \ldots, n\}$, then in addition to the above, sample m'' at random in $\mathcal{M}$.
  - Modify Step 4 and Step 5 of FMT.Encrypt$'$ as follows:
    If $s \in \{1, \ldots, n\}$, then in addition to the above, set $m'' := m'$.

  The cumulative effect of these changes is the following: if $t \neq s$, then the encrypted message m'' is random in $\mathcal{M}$; otherwise, the encrypted message m'' equals m.
- $\mathcal{H}_3$: SimExperiment. (Compute c via FMT.SimEncrypt; that is, instead of computing the proof π_{E} honestly, simulate it via NIZK.Simulate.)

Observe that $\mathcal{H}_0$ and $\mathcal{H}_1$ are indistinguishable because the distribution of t is identical in the two cases: t is uniform in $\{1, \ldots, n\}$. Also observe that $\mathcal{H}_2$ and $\mathcal{H}_3$ differ only in how the proof π_{E} is computed: in $\mathcal{H}_2$ the proof π_{E} is computed honestly, while in $\mathcal{H}_3$ the proof π_{E} is computed via NIZK.Simulate. These proofs are indistinguishable by the security of the NIZK scheme, and therefore $\mathcal{H}_2$ and $\mathcal{H}_3$ are indistinguishable. So we are left to prove that $\mathcal{H}_1$ and $\mathcal{H}_2$ are indistinguishable, which we now argue.

First, consider the case when $s \notin \{1, \ldots, n\}$. For this case, since FMT.Encrypt always samples t uniformly in $\{1, \ldots, n\}$, the ciphertext c will never decrypt to the original message m. To simulate this, whenever $s \notin \{1, \ldots, n\}$, FMT.SimEncrypt samples t uniformly in $\{1, \ldots, n\}$ and replaces m' by m''. Hence the adversary cannot distinguish between $\mathcal{H}_1$ and $\mathcal{H}_2$ by dishonestly setting s outside the range $\{1, \ldots, n\}$.

Next, consider the case when $s \in \{1, \ldots, n\}$. For this case, we prove indistinguishability by contradiction: if there exists an efficient adversary $\mathcal{A}$ that distinguishes these two hybrids then there exists an efficient adversary $\mathcal{B}$ that breaks the DDH assumption for GroupGen. This latter adversary $\mathcal{B}$ receives a DDH challenge and works as follows.

- $\mathcal{B}(\mathbb{G}, g, g^a, g^b, x)$:
  1. Compute $(\mathsf{crs}_\mathsf{K}, \mathsf{td}_\mathsf{K}) \leftarrow \mathsf{NIZK.SimSetup}(1^\lambda, \mathcal{R}_\mathsf{K}, \mathsf{Ext})$.
  2. Compute $(\mathsf{crs}_\mathsf{E}, \mathsf{td}_\mathsf{E}) \leftarrow \mathsf{NIZK.SimSetup}(1^\lambda, \mathcal{R}_\mathsf{E}, \mathsf{Sim})$.
  3. Set $g_0 := g^a$.
  4. Set $\mathsf{pp}_\mathsf{FMT} := (\mathbb{G}, q, g, g_0, \mathsf{crs}_\mathsf{K}, \mathsf{crs}_\mathsf{E})$.
  5. Give pp_FMT to $\mathcal{A}$ and obtain $(\mathsf{pk}_\mathsf{FMT}, m)$.
  6. Parse pk_FMT as $(1/n, h, \pi_\mathsf{K})$.
  7. Compute $(s, \alpha) \leftarrow \mathsf{NIZK.Extract}(\mathsf{crs}_\mathsf{K}, \mathsf{td}_\mathsf{K}, (h, g_0, g), \pi_\mathsf{K})$.
  8. Sample t at random in $\{1, \ldots, n\}$.
  9. Sample r at random in $\mathbb{Z}_q$.
  10. If $s = t$: compute $(c_1, c_2) = (g^r, m \cdot (g^\alpha)^r)$.
  11. If $s \neq t$: compute $(c_1, c_2) = (g^b, m \cdot x^t x^{-s} (g^b)^\alpha)$.
  12. Set $\pi_\mathsf{E} := \mathsf{NIZK.Simulate}(\mathsf{crs}_\mathsf{E}, \mathsf{td}_\mathsf{E}, (c_1, c_2, g, {g_0}^t h))$.
  13. Set $c := (t, c_1, c_2, \pi_\mathsf{E})$.
  14. Output $\mathcal{A}(c)$.

Now let us discuss the two cases for x.

1. *Case 1: x equals g^{ab}.* Then $c_2 = m \cdot g^{ab(t-s)} g^{b\alpha} = (g_0^t h)^b$, and c is distributed as in $\mathcal{H}_1$.

2. *Case 2: x is random in $\mathbb{G}$.* Then $c_2 = m \cdot x^{t-s} g^{b\alpha}$; letting $x = g^{ab+z}$ for some $z \in \mathbb{Z}_q$, we see that

$$\begin{aligned} c_2 &= m \cdot g^{z(t-s)} g_0^{b(t-s)} g^{b\alpha} \\ &= m' \cdot (g_0^t h)^b. \end{aligned}$$

   Here $m' = m \cdot g^{z(t-s)}$. Since x is random in $\mathbb{G}$, so is m'. Therefore c is distributed as in $\mathcal{H}_2$.

We deduce that if $\mathcal{A}$ can distinguish $\mathcal{H}_1$ and $\mathcal{H}_2$ then $\mathcal{B}$ breaks DDH. We conclude that hybrids $\mathcal{H}_0$ and $\mathcal{H}_3$ (i.e. the outputs of $\mathsf{FMT.Encrypt}$ and $\mathsf{SimExperiment}$) are indistinguishable, and therefore the output of the adversary $\mathcal{A}_\mathsf{FH}$ in the two experiments is indistinguishable, as required. □

Fractional binding. Fractional binding of our construction is implied by the following claim.

Claim A.8. *For every efficiency adversary $\mathcal{A}_\mathsf{FB}$ and transfer probability $p \in \mathcal{P}$, the following two distribution ensembles are computationally indistinguishable:*

$$\left\{ (\mathsf{pp}_\mathsf{FMT}, \mathsf{pk}_\mathsf{FMT}, \mathsf{sk}_\mathsf{FMT}, c, m) \;\middle|\; \begin{array}{r} \mathsf{pp}_\mathsf{FMT} \leftarrow \mathsf{FMT.Setup}(1^\lambda) \\ (\mathsf{pk}_\mathsf{FMT}, \mathsf{sk}_\mathsf{FMT}) \leftarrow \mathsf{FMT.Keygen}(\mathsf{pp}_\mathsf{FMT}, p) \\ c \leftarrow \mathcal{A}_\mathsf{FB}(\mathsf{pp}_\mathsf{FMT}, \mathsf{pk}_\mathsf{FMT}) \\ m \leftarrow \mathsf{FMT.Decrypt}(\mathsf{pp}_\mathsf{FMT}, \mathsf{sk}_\mathsf{FMT}, c) \end{array} \right\}_{\lambda \in \mathbb{N}} \quad \textit{and}$$

$$\left\{ (\mathsf{pp}_\mathsf{FMT}, \mathsf{pk}_\mathsf{FMT}, \mathsf{sk}'_\mathsf{FMT}, c, m') \;\middle|\; \begin{array}{r} \textit{with probability } p \textit{ set } b := 1; \textit{ else set } b := 0 \\ (\mathsf{pp}_\mathsf{FMT}, \mathsf{td}_\mathsf{FMT}) \leftarrow \mathsf{FMT.SimSetup}(1^\lambda) \\ (\mathsf{pk}_\mathsf{FMT}, \mathsf{sk}_\mathsf{FMT}) \leftarrow \mathsf{FMT.SimKeygen}(\mathsf{pp}_\mathsf{FMT}, \mathsf{td}_\mathsf{FMT}, p) \\ c \leftarrow \mathcal{A}_\mathsf{FB}(\mathsf{pp}_\mathsf{FMT}, \mathsf{pk}_\mathsf{FMT}) \\ m \leftarrow \mathsf{FMT.ExtDecrypt}(\mathsf{pp}_\mathsf{FMT}, \mathsf{td}_\mathsf{FMT}, \mathsf{sk}_\mathsf{FMT}, c) \\ \mathsf{sk}'_\mathsf{FMT} \leftarrow \mathsf{FMT.SimDecrypt}(\mathsf{pp}_\mathsf{FMT}, \mathsf{td}_\mathsf{FMT}, \mathsf{sk}_\mathsf{FMT}, b) \\ \textit{if } b = 1, \textit{ set } m' := m; \textit{ else set } m' := \varnothing \end{array} \right\}_{\lambda \in \mathbb{N}}.$$

In both experiments above, if c is not a valid ciphertext (produced by $\mathcal{A}_\mathsf{FB}$), then abort.

Proof sketch. By Claim A.5, the public parameters used in either experiment are computationally indistinguishable. In addition, by Claim A.6, the public keys used in either experiment are computationally indistinguishable. Therefore, the challenge ciphertexts c output by the adversary $\mathcal{A}_\mathsf{FB}$ in either experiment must also be computationally indistinguishable.

Now, if c is a valid ciphertext, it defines a corresponding unique pair (m, r) (since $c_1 = g^r$ binds r perfectly), and hence the output of $\mathsf{NIZK.Extract}$ equals the actual (m, r) implicitly used by the adversary under the challenge ciphertext. Next, the simulated secret key $\mathsf{sk}'_{\mathsf{FMT}}$ output by $\mathsf{sk}_{\mathsf{FMT}}$ is distributed identically to the actual $\mathsf{sk}_{\mathsf{FMT}}$ of the first experiment: when $b = 1$, it is the actual secret key, and when $b = 0$, it is a secret key having the same h as $\mathsf{sk}_{\mathsf{FMT}}$, but different s' and α' subject to the constraint that $h = g_0^{-s'} g^{\alpha'}$. The claim follows from these two points and from the indistinguishability of the ciphertexts in the two experiments. □

A.3 Definition of a fractional message transfer protocol

We define the notion of an FMT protocol by using the real/ideal world paradigm, where we compare the adversary's view in the real execution of the protocol to the output of a corresponding simulator interacting with a trusted third party that embodies the "FMT ideal functionality". Security then requires that for every efficient adversary there exists an efficient simulator such that the adversary's view (in the real world) and the simulator's output (in the ideal world) are computationally indistinguishable. Details follow.

Execution in the real world. We denote by $S_{\mathtt{R}}$ and $R_{\mathtt{R}}$ the real sender and the real receiver; the sender's input is a transfer probability p and a message m, while the receiver's input is the (same) transfer probability p. The sender and receiver interact using the FMT protocol, at the end of which the receiver learns either (i) the message m (indicating that the message transfer did occur), (ii) the distinguished message $\varnothing$ (indicating that the message transfer did not occur), or (iii) the error symbol $\perp$ (indicating that something went wrong). After the protocol, each party produces an output; together these form the output of the real execution, which we denote by $\mathtt{Real}[S_{\mathtt{R}}, R_{\mathtt{R}}](p, m)$.

Execution in the ideal world. We denote by $S_{\mathtt{I}}$ and $R_{\mathtt{I}}$ the ideal sender and the ideal receiver; the sender's input and the receiver's input are the same as in the execution in the real world. The sender and receiver interact with the ideal functionality as follows:

(1) The sender sends to the ideal functionality a transfer probability p_S, a message m_S, and a bit b_S; the bit tells the ideal functionality whether the transfer should result in error or not.

(2) The receiver sends to the ideal functionality a transfer probability p_R.

(3) If $b_S = 0$ or $p_S \neq p_R$, then the ideal functionality sends $\perp$ to the receiver. If $b_S = 1$ and $p_S = p_R$, then the ideal functionality with probability p_S sends m_S to the receiver and with probability $1 - p_S$ sends $\varnothing$ to the receiver.

Note that p_S and p_R may differ from p, and m_S may differ from m. After the above, each party produces an output; together these form the output of the ideal execution, which we denote by $\mathtt{Ideal}[S_{\mathtt{I}}, R_{\mathtt{I}}](p, m)$.

Security. An FMT protocol is *secure* if the following two conditions hold.

- *Sender security.* For every efficient receiver adversary $\tilde{R}$ in the real world there exists an efficient receiver simulator $\mathsf{Sim}_{\tilde{R}}$ in the ideal world such that, for every transfer probability p and message m, $\mathtt{Real}[S_{\mathtt{R}}, \tilde{R}](p, m)$ and $\mathtt{Ideal}[S_{\mathtt{I}}, \mathsf{Sim}_{\tilde{R}}](p, m)$ are computationally indistinguishable.

- *Receiver security.* For every efficient sender adversary $\tilde{S}$ in the real world there exists an efficient sender simulator $\mathsf{Sim}_{\tilde{S}}$ in the ideal world such that, for every transfer probability p and message m, $\mathtt{Real}[\tilde{S}, R_{\mathtt{R}}](p, m)$ and $\mathtt{Ideal}[\mathsf{Sim}_{\tilde{S}}, R_{\mathtt{I}}](p, m)$ are computationally indistinguishable.

Remark A.9. For simplicity, the above discussion does not mention the parties' auxiliary inputs. Extending the discussion to do so is straightforward. (Security with respect to auxiliary inputs implies, e.g., sequential composition.)

A.4 Construction of a fractional message transfer protocol

We construct a 2-message FMT protocol from any FMT scheme, in the common reference string model. Let $\mathsf{FMT} = (\mathsf{FMT.Setup}, \mathsf{FMT.Keygen}, \mathsf{FMT.Encrypt}, \mathsf{FMT.Decrypt})$ be an FMT scheme; the common reference string consists of $\mathsf{pp}_{\mathsf{FMT}}$ sampled according to $\mathsf{FMT.Setup}(1^\lambda)$. The protocol proceeds as follows:

- **Step 0:** The sender's input is a transfer probability p and a message m, while the receiver's input is the (same) transfer probability p.
- **Step 1 (first message):** The receiver samples a key pair $(\mathsf{pk}_{\mathsf{FMT}}, \mathsf{sk}_{\mathsf{FMT}}) \leftarrow \mathsf{FMT.Keygen}(\mathsf{pp}_{\mathsf{FMT}}, p)$ and then sends the public key $\mathsf{pk}_{\mathsf{FMT}}$ to the sender.
- **Step 2 (second message):** If $\mathsf{pk}_{\mathsf{FMT}}$ is a valid public key consistent with the public parameters $\mathsf{pp}_{\mathsf{FMT}}$ and the transfer probability p, then the sender encrypts the message m to obtain a ciphertext $c \leftarrow \mathsf{FMT.Encrypt}(\mathsf{pp}_{\mathsf{FMT}}, \mathsf{pk}_{\mathsf{FMT}}, m)$ and then sends c to the receiver. (Otherwise, the sender aborts.)
- **Step 3:** If c is a valid ciphertext, then the receiver decrypts c to obtain the message $m' \leftarrow \mathsf{FMT.Decrypt}(\mathsf{pp}_{\mathsf{FMT}}, \mathsf{sk}_{\mathsf{FMT}}, c)$, and outputs m'. (Otherwise, the receiver aborts.)

The following statement is not hard to prove.

Theorem A.10. *If* FMT *is an FMT scheme then the above is an FMT protocol (in the common reference string model).*

Proof sketch. Sender security and receiver security of the FMT protocol directly follow from the fractional hiding and fractional binding properties of the FMT scheme, respectively. Below we sketch the proofs for these two implications.

Sender security. For every efficient receiver adversary $\tilde{R}$ (in the real world) we construct an efficient receiver simulator $\mathsf{Sim}_{\tilde{R}}$ (in the ideal world), as follows.

- $\mathsf{Sim}_{\tilde{R}}$ (interacting with $\tilde{R}$ as if in the real world)
  1. Sample $(\mathsf{pp}_{\mathsf{FMT}}, \mathsf{td}_{\mathsf{FMT}}) \leftarrow \mathsf{FMT.SimSetup}(1^\lambda)$ and send $\mathsf{pp}_{\mathsf{FMT}}$ to $\tilde{R}$.
  2. The receiver adversary $\tilde{R}$ replies with $\mathsf{pk}_{\mathsf{FMT}}$. (If $\mathsf{pk}_{\mathsf{FMT}}$ is not a valid public key or is not consistent with $\mathsf{Sim}_{\tilde{R}}$'s input p, send $p_R \neq p$ to the ideal functionality and abort.)
  3. Send p to the ideal functionality.
  4. Receive a message m or $\varnothing$ from the ideal functionality.
  5. If the received message is m, then send $\mathsf{FMT.SimEncrypt}(\mathsf{pp}_{\mathsf{FMT}}, \mathsf{td}_{\mathsf{FMT}}, \mathsf{pk}_{\mathsf{FMT}}, 1, m)$ to $\tilde{R}$.
     Otherwise, sample a message m' in $\mathcal{M}$ at random and send $\mathsf{FMT.SimEncrypt}(\mathsf{pp}_{\mathsf{FMT}}, \mathsf{td}_{\mathsf{FMT}}, \mathsf{pk}_{\mathsf{FMT}}, 0, m')$ to $\tilde{R}$.
  6. Output whatever the receiver adversary $\tilde{R}$ outputs.

Security is guaranteed by the fractional hiding property of the FMT scheme:

– First, the output of $\tilde{R}$ is the output of $\mathcal{A}_{\mathsf{FH}}$ in the real world, and the output of $\mathsf{Sim}_{\tilde{R}}$ is the output of $\mathcal{A}_{\mathsf{FH}}$ in the ideal world. Since $\mathsf{Output}\,(\mathtt{Real}(\mathcal{A}_{\mathsf{FH}}))$ and $\mathsf{Output}\,(\mathtt{Ideal}(\mathcal{A}_{\mathsf{FH}}))$ are computationally indistinguishable, the outputs of $\tilde{R}$ and $\mathsf{Sim}_{\tilde{R}}$ are also computationally indistinguishable.

– Second, the outputs of $S_{\mathtt{R}}$ and $S_{\mathtt{I}}$ are both either abort or nothing. Since the probability of aborting is computationally indistinguishable in $\mathtt{Real}(\mathcal{A}_{\mathsf{FH}})$ and $\mathtt{Ideal}(\mathcal{A}_{\mathsf{FH}})$, the outputs of $S_{\mathtt{R}}$ and $S_{\mathtt{I}}$ are also computationally indistinguishable.

Receiver security. For every efficient sender adversary $\tilde{S}$ (in the real world) we construct an efficient sender simulator $\mathsf{Sim}_{\tilde{S}}$ (in the ideal world), as follows.

- $\mathsf{Sim}_{\tilde{S}}$ (interacting with $\tilde{S}$ as if in the real world)
  1. Sample $(\mathsf{pp}_{\mathsf{FMT}}, \mathsf{td}_{\mathsf{FMT}}) \leftarrow \mathsf{FMT.SimSetup}(1^\lambda)$ and send $\mathsf{pp}_{\mathsf{FMT}}$ to $\tilde{S}$.
  2. Sample $(\mathsf{pk}_{\mathsf{FMT}}, \mathsf{sk}_{\mathsf{FMT}}) \leftarrow \mathsf{FMT.SimKeygen}(\mathsf{pp}_{\mathsf{FMT}}, \mathsf{td}_{\mathsf{FMT}}, p)$ and send $\mathsf{pk}_{\mathsf{FMT}}$ to $\tilde{S}$.
  3. The sender adversary $\tilde{S}$ replies with a ciphertext c. (If c is an invalid ciphertext, send to the ideal functionality p, a random message m_S in $\mathcal{M}$, and $b_S = 0$, and abort.)
  4. Compute $(m) \leftarrow \mathsf{FMT.ExtDecrypt}(\mathsf{pp}_{\mathsf{FMT}}, \mathsf{td}_{\mathsf{FMT}}, \mathsf{sk}_{\mathsf{FMT}}, c)$ and send $(p, m, b_S = 1)$ to the ideal functionality.
  5. Output whatever the sender adversary $\tilde{S}$ outputs.

Security is guaranteed by the fractional binding property of the FMT scheme:

– First, the probability of aborting in $\mathtt{Real}(\mathcal{A}_{\mathsf{FB}})$ and in $\mathtt{Ideal}(\mathcal{A}_{\mathsf{FB}})$ are computationally indistinguishable, so the view of $\tilde{S}$ in the real world is computationally indistinguishable from its view when interacting with $\mathsf{Sim}_{\tilde{S}}$. Therefore the output of $\tilde{S}$ and output of $\mathsf{Sim}_{\tilde{S}}$ are also computationally indistinguishable.

– Second, the output of $R_{\mathtt{R}}$ is the output of $\mathtt{Real}(\mathcal{A}_{\mathsf{FB}})$, and the output of $R_{\mathtt{I}}$ is the output of $\mathtt{Ideal}(\mathcal{A}_{\mathsf{FB}})$. Since $\mathsf{Output}\,(\mathtt{Real}(\mathcal{A}_{\mathsf{FB}}))$ and $\mathsf{Output}\,(\mathtt{Ideal}(\mathcal{A}_{\mathsf{FB}}))$ are computationally indistinguishable, the outputs of $R_{\mathtt{R}}$ and $R_{\mathtt{I}}$ are also computationally indistinguishable. □

B Security of decentralized anonymous payment systems

We use DAP schemes that generalize in a straightforward way those in [BCG+14, GGM16] (see Section 5.4); moreover, we rely on a *simulation-based* security definition, stronger than previous game-based ones. The purpose of this section is to specify this security definition. We introduce auxiliary algorithms and notions (Appendix B.1) and then specify the security definition via real and ideal experiments (Appendix B.2); afterwards, we comment on suitable instantiations (Appendix B.3).

B.1 Auxiliary algorithms and notions

Auxiliary algorithms for a DAP scheme. A secure DAP scheme possesses the following auxiliary algorithms, some of which have oracle access to the ledger $\mathbf{L}$:

$$(\mathsf{DAP.SimSetup}, \mathsf{DAP.SimPour}^{\mathbf{L}}, \mathsf{DAP.SimMint}^{\mathbf{L}}, \mathsf{DAP.ExtractMint}^{\mathbf{L}}, \mathsf{DAP.ExtractPour}^{\mathbf{L}})$$

These are defined as follows:

- $\underline{\mathsf{DAP.SimSetup}(1^\lambda, \Pi_\mathsf{m}, \Pi_\mathsf{p}) \to (\mathsf{pp}_\mathsf{DAP}, \mathsf{td}_\mathsf{DAP}).}$

  On input a security parameter 1^λ, mint predicate Π_m, and pour predicate Π_p, DAP.SimSetup outputs simulated DAP parameters pp_DAP and a trapdoor td_DAP.

- $\underline{\mathsf{DAP.SimMint}^{\mathbf{L}}(\mathsf{pp}_\mathsf{DAP}, \mathsf{td}_\mathsf{DAP}, v, \mathsf{pub}) \to (\mathbf{c}, \mathsf{tx}_\mathsf{m}).}$

  On input simulated public parameters pp_DAP, trapdoor td_DAP, new coin value v, and public information string pub, and with oracle access to $\mathbf{L}$, $\mathsf{DAP.SimMint}^{\mathbf{L}}$ outputs a new coin $\mathbf{c}$ and a simulated mint transaction tx_m that is consistent with $\mathbf{c}$.

- $\underline{\mathsf{DAP.SimPour}^{\mathbf{L}}(\mathsf{pp}_\mathsf{DAP}, \mathsf{td}_\mathsf{DAP}, m, n, v_\mathsf{pub}, \Delta, [\mathbf{c}_k]_1^p) \to ([\mathbf{c}_j]_1^n, \mathsf{tx}_\mathsf{p}).}$

  On input simulated public parameters pp_DAP, trapdoor td_DAP, number of input coins m, number of output coins n, public value v_pub, activation delay Δ, and $0 \le p \le n$ output coins $[\mathbf{c}_k]_1^p$, and with oracle access to $\mathbf{L}$, DAP.SimPour outputs new coins $[\mathbf{c}_j]_1^n$ that contain $[\mathbf{c}_k]_1^p$, and a simulated pour transaction tx_p that has activation delay Δ and is consistent with $[\mathbf{c}_j]_1^n$.

- $\underline{\mathsf{DAP.ExtractMint}^{\mathbf{L}}(\mathsf{pp}_\mathsf{DAP}, \mathsf{td}_\mathsf{DAP}, \mathsf{tx}_\mathsf{m}) \to (v, \mathsf{apk}, \mathsf{pub}, \mathsf{sec}, \mathbf{c}).}$

  On input simulated public parameters pp_DAP, trapdoor td_DAP, and a mint transaction tx_m, and with oracle access to $\mathbf{L}$, DAP.ExtractMint extracts the inputs used to create tx_m, along with the generated new coin.

- $\underline{\mathsf{DAP.ExtractPour}^{\mathbf{L}}(\mathsf{pp}_\mathsf{DAP}, \mathsf{td}_\mathsf{DAP}, \mathsf{tx}_\mathsf{p}) \to ([\mathbf{c}_i]_1^m, [\mathsf{ask}_i]_1^m, [v_j]_1^n, [\mathsf{apk}_j]_1^n, [\mathsf{pub}_j]_1^n, [\mathsf{sec}_j]_1^n, v_\mathsf{pub}, \mathsf{info}, \Delta, [\mathbf{c}_j]_1^n).}$

  On input simulated public parameters pp_DAP, trapdoor td_DAP, and a pour transaction tx_p, and with oracle access to $\mathbf{L}$, DAP.ExtractPour extracts the inputs used to create tx_p, along with the generated new coins.

The algorithms DAP.ExtractMint and DAP.ExtractPour are required to work even when the adversary has received simulated outputs from DAP.SimMint and DAP.SimPour. This definition is analogous to *simulation extractability* and we formalize it in our definitions further below.

Consistent transcripts. We introduce the notion of a *transcript*, and then define what it means for a transcript to be *consistent*. A transcript $\mathbf{T}$ is a list of transcript entries te, each of two possible types:

1. *Mint entries (denoted* te_m*).* A mint entry is a tuple of the form $\mathsf{te}_\mathsf{m} := (v, \mathsf{apk}, \mathsf{pub}, \mathsf{sec}, \mathbf{c}, \mathsf{tx}_\mathsf{m})$. Thus a mint entry consists of the inputs and outputs of an invocation of DAP.Mint.

2. *Pour entries (denoted* te_p*).* A pour entry is a tuple of the form $\mathsf{te}_\mathsf{p} := ([\mathbf{c}_i]_1^m, [\mathsf{ask}_i]_1^m, [v_j]_1^n, [\mathsf{apk}_j]_1^n, [\mathsf{pub}_j]_1^n, [\mathsf{sec}_j]_1^n, v_\mathsf{pub}, \mathsf{info}, \Delta, [\mathbf{c}_j]_1^n, \mathsf{tx}_\mathsf{p})$. Thus a pour entry consists of the inputs and outputs of an invocation of DAP.Pour.

A transcript entry is *active* if the contained transaction has become active on the ledger $\mathbf{L}$. Otherwise, the entry is *dormant*. (For definitions of active and dormant transactions, see Sections 5.1 and 5.4.)

Definition B.1. *A transcript* $\mathbf{T}$ *is* **consistent** *if each transcript entry is consistent as defined below.*

1. *A* mint entry $\mathsf{te} = (v, \mathsf{apk}, \mathsf{pub}, \mathsf{sec}, \mathbf{c}, \mathsf{tx}_\mathsf{m})$ *is consistent if the mint predicate evaluated on the mint inputs outputs* 1*, that is,* $\Pi_\mathsf{m}(v, \mathsf{apk}, \mathsf{pub}, \mathsf{sec}) = 1$.

2. *A* pour entry $\mathsf{te} = ([\mathbf{c}_i]_1^m, [\mathsf{ask}_i]_1^m, [v_j]_1^n, [\mathsf{apk}_j]_1^n, [\mathsf{pub}_j]_1^n, [\mathsf{sec}_j]_1^n, v_\mathsf{pub}, \mathsf{info}, \Delta, [\mathbf{c}_j]_1^n, \mathsf{tx}_\mathsf{p})$ *is valid if the following conditions are satisfied.*

   (a) *The pour predicate evaluated on the pour inputs outputs* 1*, that is,*

   $$\Pi_\mathsf{p}([\mathbf{c}_i]_1^m, [\mathsf{ask}_i]_1^m, [v_j]_1^n, [\mathsf{apk}_j]_1^n, [\mathsf{pub}_j]_1^n, [\mathsf{sec}_j]_1^n, v_\mathsf{pub}, \mathsf{info}, \Delta) = 1 \ .$$

   (b) *The commitment of each input coin in* $[\mathbf{c}_i]_1^m$ *appears exactly once in some previous* active *transcript entry.*
   (c) *The total sum of coin values in* $[\mathbf{c}_j]_1^n$ *plus the public output* v_pub *is equal to the sum of the values in* $[\mathbf{c}_i]_1^m$*, that is,* $\sum_{i=1}^m \mathbf{c}_i.v = \sum_{j=1}^n v_j + v_\mathsf{pub}$.
   (d) *The secret key* ask_i *for an input coin* $\mathbf{c}_i$ *corresponds to* $\mathbf{c}_i$*'s public key* apk_i *that appears in a previous transcript entry that specifies* $\mathbf{c}_i$*'s creation.*

   With these preliminaries in place, we now provide our main security definition for a DAP scheme.

B.2 Security definition

We define security of a DAP scheme by requiring indistinguishability of any efficient adversary's outcome in two experiments. In Appendix B.2.1, we define the real-world experiment, and in Appendix B.2.2 we define the ideal-world experiment. Finally in Appendix B.2.3, we provide the formal security definition of a DAP scheme with respect to these two experiments.

In both experiments, we consider an adversary $\mathcal{A}$ that interacts with a challenger. The challenger maintains two tables: a *coin table* Coins and an *address table* Addr. The experiments are parameterized by a security parameter λ and two predicates $\Pi_\mathsf{m}, \Pi_\mathsf{p}$. In each experiment a ledger $\mathbf{L}$ and a transcript $\mathbf{T}$ are first initialized to be empty. The adversary $\mathcal{A}$ can read transactions from and append transactions to $\mathbf{L}$, with the caveat that any appended transaction tx must satisfy $\mathsf{DAP.VerifyTransaction}^{\mathbf{L}}(\mathsf{pp}_\mathsf{DAP}, \mathsf{tx}) = 1$.

B.2.1 Real experiment

At the beginning of the real experiment (denoted REAL), $\mathcal{A}$ is given honestly-generated public parameters $\mathsf{pp}_\mathsf{DAP} \leftarrow \mathsf{DAP.Setup}(1^\lambda, \Pi_\mathsf{m}, \Pi_\mathsf{p})$, and can subsequently issue to the challenger the following queries sequentially and in any order:

- *Creating addresses:* $\mathsf{query} = (\texttt{CreateAddr}, \mathsf{meta})$.

  1. Compute $(\mathsf{apk}, \mathsf{ask}) \leftarrow \mathsf{DAP.CreateAddr}(\mathsf{pp}_\mathsf{DAP}, \mathsf{meta})$.
  2. Set $\mathsf{Addr}[\mathsf{apk}] := \mathsf{ask}$.
  3. Return apk to $\mathcal{A}$.

- *Minting coins:* $\mathsf{query} = (\texttt{Mint}, (v, \mathsf{apk}, \mathsf{pub}, \mathsf{sec}))$.

  1. Compute $(\mathbf{c}, \mathsf{tx}_\mathsf{m}) \leftarrow \mathsf{DAP.Mint}^{\mathbf{L}}(v, \mathsf{apk}, \mathsf{pub}, \mathsf{sec})$.
  2. If DAP.Mint returns $\perp$: halt and return $\perp$ to $\mathcal{A}$. Otherwise, write tx_m to $\mathbf{L}$.
  3. Construct a transcript entry $\mathsf{te}_\mathsf{m} := (v, \mathsf{apk}, \mathsf{pub}, \mathsf{sec}, \mathbf{c}, \mathsf{tx}_\mathsf{m})$ and append te_m to $\mathbf{T}$.
  4. Parse tx_m to obtain cm.
  5. Set $\mathsf{Coins}[\mathsf{cm}] := (\mathbf{c}, \mathsf{tx}_\mathsf{m}, 0)$ and clk in the coin table.

- *Pouring coins:* $\mathsf{query} = (\texttt{Pour}, ([\mathsf{cm}_i]_1^m, [v_j]_1^n, [\mathsf{apk}_j]_1^n, [\mathsf{pub}_j]_1^n, [\mathsf{sec}_j]_1^n, \Delta, v_\mathsf{pub}, \mathsf{info}))$.

1. For $i = 1$ to m:
   (a) if $\mathsf{Coins}[\mathsf{cm}_i] = \perp$: return $\perp$ to $\mathcal{A}$.
   (b) retrieve $(\mathbf{c}_i, \mathsf{tx}_i, \Delta_i) := \mathsf{Coins}[\mathsf{cm}_i]$.
   (c) if tx_i is not active on the ledger: return $\perp$ to $\mathcal{A}$.
   (d) if $\mathsf{Addr}[\mathsf{apk}] = \perp$: return $\perp$ to $\mathcal{A}$.
   (e) retrieve $\mathsf{ask}_i := \mathsf{Addr}[\mathbf{c}_i.\mathsf{apk}]$.
2. Use the retrieved coins and secret keys to construct $[\mathbf{c}_i]_1^m$ and $[\mathsf{ask}_i]_1^m$.
3. Compute $([\mathbf{c}_j]_1^n, \mathsf{tx}_\mathsf{p}) \leftarrow \mathsf{DAP.Pour}^{\mathbf{L}}(\mathsf{pp}_\mathsf{DAP}, [\mathbf{c}_i]_1^m, [\mathsf{ask}_i]_1^m, [v_j]_1^n, [\mathsf{apk}_j]_1^n, [\mathsf{pub}_j]_1^n, [\mathsf{sec}_j]_1^n, v_\mathsf{pub}, \mathsf{info}, \Delta)$.
4. If DAP.Pour returns $\perp$: return $\perp$ to $\mathcal{A}$.
5. Parse tx_p as $(\mathsf{ts}, \Delta, [\mathsf{sn}_i]_1^m, [\mathsf{cm}_j]_1^n, v_\mathsf{pub}, \mathsf{info}, *)$.
6. For $j = 1$ to n: set $\mathsf{Coins}[\mathsf{cm}_j] := (\mathbf{c}_j, \mathsf{tx}_\mathsf{p}, \Delta)$.
7. Construct a transcript entry $\mathsf{te}_\mathsf{p} := ([\mathbf{c}_i]_1^m, [\mathsf{ask}_i]_1^m, [v_j]_1^n, [\mathsf{apk}_j]_1^n, [\mathsf{pub}_j]_1^n, [\mathsf{sec}_j]_1^n, v_\mathsf{pub}, \mathsf{info}, \Delta, [\mathbf{c}_j]_1^n, \mathsf{tx}_\mathsf{p})$.
8. Append te_p to $\mathbf{T}$.
9. Append tx_p to $\mathbf{L}$.

- *Receiving coins:* query $= (\texttt{Receive})$.

1. For all apk such that $\mathsf{Addr}[\mathsf{apk}] \neq \perp$:
   (a) retrieve $\mathsf{ask} := \mathsf{Addr}[\mathsf{apk}]$.
   (b) compute $[\mathbf{c}_j]_1^n \leftarrow \mathsf{DAP.Receive}^{\mathbf{L}}(\mathsf{pp}_\mathsf{DAP}, (\mathsf{apk}, \mathsf{ask}))$.
   (c) For $j = 1$ to n:
      i. retrieve the transaction tx containing $\mathbf{c}_j.\mathsf{cm}$ from $\mathbf{L}$.
      ii. set $\mathsf{Coins}[\mathbf{c}_j.\mathsf{cm}] := (\mathbf{c}_j, \mathsf{tx}, \mathsf{tx}.\Delta)$.

B.2.2 Ideal experiment

The ideal experiment (denoted IDEAL) proceeds similarly to REAL, but with the following modifications. At the beginning, the challenger computes simulated parameters and a trapdoor: $(\mathsf{pp}_\mathsf{DAP}, \mathsf{td}_\mathsf{DAP}) \leftarrow \mathsf{DAP.SimSetup}(1^\lambda)$, and gives the simulated parameters to the adversary $\mathcal{A}$ in place of the honestly-generated parameters. The `CreateAddr` and `Receive` queries are handled identically to the REAL experiment, but the `Mint` and `Pour` queries are handled as follows:

Minting coins: query $= (\texttt{Mint}, (v, \mathsf{apk}, \mathsf{pub}, \mathsf{sec}))$.

1. If $\Pi_\mathsf{m}(v, \mathsf{apk}, \mathsf{pub}, \mathsf{sec}) = 0$: abort and return $\perp$ to $\mathcal{A}$.
2. Compute $(\mathbf{c}, \mathsf{tx}_\mathsf{m}) \leftarrow \mathsf{DAP.SimMint}^{\mathbf{L}}(\mathsf{pp}_\mathsf{DAP}, \mathsf{td}_\mathsf{DAP}, v, \mathsf{pub})$.
3. If DAP.SimMint returns $\perp$: abort and output $\perp$ to $\mathcal{A}$. Otherwise write tx_m to $\mathbf{L}$.
4. Construct a transcript entry $\mathsf{te}_\mathsf{m} := (v, \mathsf{apk}, \mathsf{pub}, \mathsf{sec}, \mathbf{c}, \mathsf{tx}_\mathsf{m})$ and append te_m to $\mathbf{T}$.
5. Set $\mathsf{Coins}[\mathbf{c}.\mathsf{cm}] := (\mathbf{c}, \mathsf{tx}_\mathsf{m}, 0)$.

Pouring coins: query $= (\texttt{Pour}, ([\mathsf{cm}_i]_1^m, [v_j]_1^n, [\mathsf{apk}_j]_1^n, [\mathsf{pub}_j]_1^n, [\mathsf{sec}_j]_1^n, v_\mathsf{pub}, \mathsf{info}, \Delta))$.

1. For $i = 1$ to m:
   (a) if $\mathsf{Coins}[\mathsf{cm}_i] = \perp$: return $\perp$ to $\mathcal{A}$.
   (b) retrieve $(\mathbf{c}_i, \mathsf{tx}_i, \Delta_i) := \mathsf{Coins}[\mathsf{cm}_i]$.
   (c) if tx_i is not active on the ledger: return $\perp$ to $\mathcal{A}$.
   (d) if $\mathsf{Addr}[\mathsf{apk}] = \perp$: return $\perp$ to $\mathcal{A}$.
   (e) retrieve $\mathsf{ask}_i := \mathsf{Addr}[\mathbf{c}_i.\mathsf{apk}]$.
2. Evaluate Π_p on all of the inputs and retrieved values. If it outputs 0, abort and return $\perp$ to $\mathcal{A}$.
3. For $j = 1$ to n: if $\mathsf{Addr}[\mathsf{apk}_j] = \perp$: $\mathbf{c}_k \leftarrow \mathsf{DAP.Mint}(\mathsf{pp}_\mathsf{DAP}, v_j, \mathsf{apk}_j, \mathsf{pub}_j, \mathsf{sec}_j)$.
4. Compute $([\mathbf{c}_j]_1^n, \mathsf{tx}_\mathsf{p}) \leftarrow \mathsf{DAP.SimPour}^{\mathbf{L}}(\mathsf{pp}_\mathsf{DAP}, \mathsf{td}_\mathsf{DAP}, m, n, v_\mathsf{pub}, \Delta, [\mathbf{c}_k]_1^p)$.
5. If DAP.SimPour returns $\perp$: abort and return $\perp$ to $\mathcal{A}$.
6. Parse tx_p as $(\mathsf{ts}, \Delta, [\mathsf{sn}_i]_1^m, [\mathsf{cm}_j]_1^n, v_\mathsf{pub}, \mathsf{info}, *)$.

7. For $j = 1$ to n: set $\mathsf{Coins}[\mathbf{c}_j.\mathsf{cm}] := (\mathbf{c}_j, \mathsf{tx}_\mathsf{p}, \Delta)$.
8. Construct a transcript entry $\mathsf{te}_\mathsf{p} := ([\mathbf{c}_i]_1^m, [\mathsf{ask}_i]_1^m, [v_j]_1^n, [\mathsf{apk}_j]_1^n, [\mathsf{pub}_j]_1^n, [\mathsf{sec}_j]_1^n, v_\mathsf{pub}, \mathsf{info}, \Delta, [\mathbf{c}_j]_1^n, \mathsf{tx}_\mathsf{p})$.
9. Append te_p to $\mathbf{T}$.
10. Append tx_p to $\mathbf{L}$.

In addition, each time $\mathcal{A}$ appends a valid transaction $\mathsf{tx} \in \{\mathsf{tx}_\mathsf{m}, \mathsf{tx}_\mathsf{p}\}$ to $\mathbf{L}$, the challenger runs $\mathsf{DAP.ExtractMint}^{\mathbf{L}}$ or $\mathsf{DAP.ExtractPour}^{\mathbf{L}}$ (respectively) and constructs a transcript entry te to append to $\mathbf{T}$. If any of the following events occur, the challenger returns `inconsistent` to $\mathcal{A}$ and halts the experiment:

1. The algorithm $\mathsf{DAP.ExtractMint}^{\mathbf{L}}$ or $\mathsf{DAP.ExtractPour}^{\mathbf{L}}$ outputs $\perp$;
2. The resulting transcript $\mathbf{T}$ becomes *inconsistent*, in the sense of Definition B.1;
3. A pour entry te_p constructed by extracting from a tx_p written to $\mathbf{L}$ by the $\mathcal{A}$ spends coins currently registered to a public key apk such that $\mathsf{Addr}[\mathsf{apk}] \neq \perp$. (Such an entry is equivalent to the adversary attempting to spend coins it does not "own".)

This concludes the description of the IDEAL experiment. Having defined this experiment, we are now ready to provide a formal definition for the security of a DAP scheme.

B.2.3 Formal security definition

Definition B.2. *We say that a DAP scheme* DAP *is* **secure** *for a security parameter λ and the predicates $(\Pi_\mathsf{m}, \Pi_\mathsf{p})$ if for every efficient adversary $\mathcal{A}$, $\mathcal{A}$'s output at the end of the* REAL *and* IDEAL *experiments is computationally indistinguishable.*

Discussion. Intuitively, we note that this definition preserves anonymity because the simulated Mint and Pour algorithms DAP.SimMint and DAP.SimPour do not receive any secret information about the coins being spent. Simultaneously, this definition implies that for every set of transactions written to $\mathbf{L}$ by $\mathcal{A}$, we can construct a valid transcript that contains all coin secrets and is *consistent*. Thus, under this definition, no adversary $\mathcal{A}$ can submit transactions that violate the financial invariant, double spend the same coin, or manufacture coins except as the output of a valid mint or pour transaction that has been pushed to $\mathbf{L}$. Moreover, these properties hold simultaneously, that is, the extraction algorithms must work correctly even when the adversary receives transactions generated using td_DAP via the DAP.SimPour or DAP.SimMint algorithms. We also note that Definition B.2 is a strictly stronger definition than the various security definitions provided by [BCG+14, Appendix C].

B.3 Security of existing DAP constructions

DAP schemes were constructed in [BCG+14], and proved secure relative to a set of game-based definitions targeting various security goals. Subsequent work identified some limitations in these game-based definitions (e.g., faerie gold [HBHW16]), and proposed not only strong simulation-based security definitions for DAP schemes [GGM16], but also simulation-based *and composable* definitions for anonymous smart contracts [KMS+16]. Our security definitions are also simulation-based, and consider a notion of a DAP scheme that extends prior ones [BCG+14, GGM16] via straightforward modifications of its interface (see Section 5.4), justified via simple modifications of the underlying construction.

One of the main cryptographic primitives used in constructions of DAP schemes is non-interactive zero knowledge proofs of knowledge. When aiming for simulation-based security definitions, these have to continue to be proofs of knowledge even given previous simulated proofs, i.e., they have to be *simulation extractable* [Sah99, DDO+01]. Indeed, the aforementioned subsequent works achieve simulation-based security definitions by 'retrofitting' DAP schemes with such stronger proof systems; we too rely on these. The auxiliary algorithms mentioned in Appendix B.1 then follow in a straightforward way from the algorithms for simulating proofs and extracting witnesses in these proof systems.

C Security of our DAM construction

In this section we complete the formalization of the security definitions for DAM, and provide a proof of security for the DAM construction presented in Section 7.3. We now begin by defining the operation of the honest parties.

The honest real and ideal-world parties. Each honest party in both the real and ideal worlds is provided with an adaptive execution strategy Σ. The execution strategy is a stateful efficient program that is run by the honest party continuously to produce a series of instructions, where each instruction is one of $(\mathsf{RegisterAddress}, \mathsf{IncrementEpoch}, \mathsf{MintCoin}, \mathsf{MintDeposit}, \mathsf{MintTicket}, \mathsf{PourCoinToDeposit}, \mathsf{PourCoinToTicket}, \mathsf{WithdrawDeposit}, \mathsf{RefreshTicket}, \mathsf{PourTicket})$. Similarly, the existence of new transactions is reported to Σ. Crucially, all instructions reference coins by public pseudonyms for coins, deposits and tickets; and no secret information is provided to Σ in response to any instruction. The distribution of these coin pseudonyms is such that in both the real and ideal worlds the distribution of inputs to Σ is structured identically.

With these building blocks in place, we now proceed to a proof of security for our DAM scheme of Section 7.3. That is, we prove Theorem 7.1, which states that our construction of a DAM scheme in Section 7.3 is secure in the sense of Definition 6.1. More formally, assuming the following, we prove that for every real-world adversary $\mathcal{A}$ against our DAM scheme construction DAM, there exists a ideal-world adversary $\mathcal{S}$ such that the outputs of the real-world and ideal-world experiments are indistinguishable.

1. All parties communicate through secure, authenticated channels;
2. In the real world, all parties have access to a trusted global append-only ledger $\mathbf{L}$;
3. DAP is a DAP scheme secure in the sense of Definition B.2;
4. FMT is an FMT scheme secure in the sense of Appendix A.1;
5. NIZK is a simulation-extractable non-interactive perfect zero knowledge argument system;
6. COMM is a commitment scheme;
7. CRH is a collision-resistant hash function;
8. PRF is a (collision-resistant) pseudorandom function;
9. SIG is a strongly-unforgeable one-time signature scheme.

Roadmap. In Section C.1, we construct $\mathcal{S}$, and then in Section C.2, we prove that the outputs of the adversary $\mathcal{A}$ in the real-world and ideal-world experiments are indistinguishable.

C.1 Ideal-world adversary $\mathcal{S}$

Let us now define $\mathcal{S}$, which runs $\mathcal{A}$ internally and interacts with T_p as described below. At the start of the experiment, $\mathcal{S}$ first initializes the following tables to be empty:

– SimNotes, which maps real notes to their ideal identifiers.
– SimARates, which maps a session identifier sid to a set of average-case rate limit tags for payments to sid.
– SimWRates, which maps a session identifier sid to a set of worst-case rate limit tags for payments to sid.
– SimTokens, which maps ideal ticket tokens to the double spend tag information for the corresponding pour-ticket transaction.
– SimPourTk, which maps pour-ticket transactions to their corresponding ideal ticket tokens.

Next, $\mathcal{S}$ runs generates simulated parameters and a trapdoor for the DAM scheme as follows:
1. $(\mathsf{pp}_\mathsf{DAP}, \mathsf{td}_\mathsf{DAP}) \leftarrow \mathsf{DAP.SimSetup}(1^\lambda, \Pi_\mathsf{m}, \Pi_\mathsf{p})$.
2. $(\mathsf{pp}_\mathsf{FMT}, \mathsf{td}_\mathsf{FMT}) \leftarrow \mathsf{FMT.SimSetup}(1^\lambda)$.
3. $\mathsf{pp}_\mathsf{SIG} \leftarrow \mathsf{SIG.Setup}(1^\lambda)$.
4. $(\mathsf{crs}_\mathsf{pt}, \mathsf{td}_\mathsf{pt}) \leftarrow \mathsf{NIZK.SimSetup}(1^\lambda, \mathcal{R}_\mathsf{pt})$.
5. $\mathsf{pp} := (\mathsf{pp}_\mathsf{DAP}, \mathsf{pp}_\mathsf{FMT}, \mathsf{pp}_\mathsf{SIG}, \mathsf{crs}_\mathsf{pt})$.
6. $\mathsf{td} := (\mathsf{td}_\mathsf{DAP}, \mathsf{td}_\mathsf{FMT}, \mathsf{td}_\mathsf{pt})$.

$\mathcal{S}$ now runs $\mathcal{A}$ internally, providing it with the simulated public parameters parameters pp. $\mathcal{S}$ initializes an empty transcript $\mathbf{T}$, simulates the ledger $\mathbf{L}$, responds to $\mathcal{A}$'s attempts to append to $\mathbf{L}$, and interacts with the trusted party T_p as follows:

1. in Section C.1.1, we detail how $\mathcal{S}$ increments the ledger epoch and which transactions it allows onto the ledger.
2. In Section C.1.2, we detail how $\mathcal{S}$ responds to actions initiated by $\mathcal{A}$, such as appending mint, pour-coin, pour-ticket, withdraw and refresh transactions to the ledger, as well as interacting with the trusted party whenever $\mathcal{A}$ wishes to engage in a PourTicket protocol (as sender) with another honest party.
3. In Section C.1.3, we detail how $\mathcal{S}$ responds to notifications from $\mathsf{T_p}$ for mints, pour-coins, pour-ticket, withdraw and refresh transactions, and also how $\mathcal{S}$ mediates between the trusted party and $\mathcal{A}$ when an honest party (acting as sender) initiates a PourTicket protocol with a corrupted party controlled by $\mathcal{A}$.

Finally, we assume that whenever $\mathcal{S}$ extracts from or simulates a transaction, it creates a corresponding transcript entry, appends it to the transcript $\mathbf{T}$, and checks that $\mathbf{T}$ is consistent as defined in Definition B.1.

C.1.1 Ledger operations

Ledger epochs. Whenever $\mathcal{A}$ attempts to increment the ledger epoch $\mathbf{L}.\mathsf{Epoch}$, $\mathcal{S}$ invokes $\mathsf{T_p}.\mathsf{IncrementEpoch}()$. If $\mathcal{S}$ receives a 'increment' message `inc` from $\mathsf{T_p}$,

Validating ledger transactions. Whenever $\mathcal{A}$ attempts to send a transaction tx to $\mathbf{L}$, $\mathcal{S}$ writes tx to $\mathbf{L}$ if and only if $\mathsf{DAM.VerifyTransaction}^{\mathbf{L}}(\mathsf{pp}, \mathsf{tx}) = 1$.

C.1.2 Adversarial actions

When $\mathcal{A}$ performs the following actions, $\mathcal{S}$ interacts with the trusted party $\mathsf{T_p}$ as follows.

Adversarial mints. When $\mathcal{A}$ posts a mint transaction $\mathsf{tx_m}$ to $\mathbf{L}$, $\mathcal{S}$ performs the following actions.

if $\mathsf{pub} = \mathtt{cn}$	if $\mathsf{pub} = \mathtt{dp}$	if $\mathsf{pub} = \mathtt{tk}$
1. $(v, \mathsf{apk}, \mathsf{pub}, \mathsf{sec}, \mathbf{n}) \leftarrow \mathsf{DAP.ExtractMint}(\mathsf{pp_{DAP}}, \mathsf{td_{DAP}}, \mathsf{tx_m})$. 2. If $\mathsf{DAP.ExtractMint}$ returns $\perp$: abort and return $\perp$ to $\mathcal{A}$.		
3. Invoke $\mathsf{T_p}.\mathsf{MintCoin}(v, \mathsf{apk})$. 4. **Receive from $\mathsf{T_p}$:** $(\mathtt{mintcn}, \mathbb{c}, v)$. 5. Set $\mathsf{SimNotes}[\mathbf{n}] := \mathbb{c}$.	3. Invoke $\mathsf{T_p}.\mathsf{MintDeposit}(v, \mathsf{apk}, \varnothing)$. 4. **Receive from $\mathsf{T_p}$:** $(\mathtt{mintdp}, \mathbb{d}, v)$. 5. Set $\mathsf{SimNotes}[\mathbf{n}] := \mathbb{d}$.	3. Parse sec as $\mathbf{d}$. 4. Retrieve $\mathbb{d} := \mathsf{SimNotes}[\mathbf{d}]$. 5. Set $\mathsf{info}^{\mathbb{d}} := (\mathbb{d}, \mathbf{d}.v, \mathbf{d}.\mathsf{apk})$. 6. Invoke $\mathsf{T_p}.\mathsf{MintTicket}(v, \mathsf{apk}, \mathsf{info}^{\mathbb{d}})$. 7. **Receive from $\mathsf{T_p}$:** $(\mathtt{minttk}, \mathbb{t}, v)$. 8. Set $\mathsf{SimNotes}[\mathbf{n}] := \mathbb{t}$.

Adversarial pour-coins. When $\mathcal{A}$ posts a pour-coin transaction $\mathsf{tx_{pc}}$ to $\mathbf{L}$, $\mathcal{S}$ performs the following actions.

1. $([\mathbf{c}_i]_1^m, [\mathsf{ask}_i]_1^m, [v_j]_1^n, [\mathsf{apk}_j]_1^n, [\mathsf{pub}_j]_1^n, [\mathsf{sec}_j]_1^n, v_{\mathsf{pub}}, \mathsf{info}, \Delta, [\mathbf{n}_j]_1^n) \leftarrow \mathsf{DAP.ExtractPour}(\mathsf{pp_{DAP}}, \mathsf{td_{DAP}}, \mathsf{tx_{pc}})$.
2. For $i = 1$ to m: retrieve $\mathbb{c}_i := \mathsf{SimNotes}[\mathbf{c}_i]$

if for all $j \in \{1, \ldots, n\}$, $\mathsf{pub}_j = \mathtt{cn}$	if for all $j \in \{1, \ldots, n\}$, $\mathsf{pub}_j = \mathtt{dp}$:
1. $\mathsf{T_p}.\mathsf{PourCoinToCoin}([\mathbb{c}_i]_1^m, [\mathsf{apk}_j]_1^n, [v_j]_1^n, v_{\mathsf{pub}})$. 2. **Receive from $\mathsf{T_p}$:** $(\mathtt{pourcn}, m, [\mathbb{c}_j]_1^n, v_{\mathsf{pub}})$. 3. for $j = 1$ to n: set $\mathsf{SimNotes}[\mathbf{n}_j] := \mathbb{c}_j$.	1. $\mathsf{T_p}.\mathsf{PourCoinToDeposit}([\mathbb{c}_i]_1^m, [\mathsf{apk}_j]_1^n, [v_j]_1^n, v_{\mathsf{pub}}, [\varnothing_j]_1^n)$. 2. **Receive from $\mathsf{T_p}$:** $(\mathtt{pourcn}, m, [\mathbb{d}_j]_1^n, v_{\mathsf{pub}})$. 3. for $j = 1$ to n: set $\mathsf{SimNotes}[\mathbf{n}_j] := \mathbb{d}_j$.
if for all $j \in \{1, \ldots, n\}$, $\mathsf{pub}_j = \mathtt{tk}$:	
1. for $j = 1$ to n: (a) parse sec_j as $\mathbf{d}_j$. (b) retrieve $\mathbb{d}_j := \mathsf{SimNotes}[\mathbf{d}_j]$. (c) $\mathsf{info}_j^{\mathbb{d}} := (\mathbb{d}_j, \mathbf{d}_j.v, \mathbf{d}_j.\mathsf{apk})$. 2. $\mathsf{T_p}.\mathsf{PourCoinToTicket}([\mathbb{c}_i]_1^m, [\mathsf{apk}_j]_1^n, [v_j]_1^n, v_{\mathsf{pub}}, [\mathsf{info}^{\mathbb{d}}{}_j]_1^n)$. 3. **Receive from $\mathsf{T_p}$:** $(\mathtt{pourcn}, m, [\mathbb{t}_j]_1^n, v_{\mathsf{pub}})$. 4. for $j = 1$ to n: set $\mathsf{SimNotes}[\mathbf{n}_j] := \mathbb{t}_j$.	

Adversarial deposit withdrawals and ticket refreshes. When $\mathcal{A}$ posts a withdraw transaction $\mathsf{tx}_{\mathsf{wd}}$ or refresh transaction $\mathsf{tx}_{\mathsf{ref}}$ to $\mathbf{L}$, $\mathcal{S}$ performs the following actions.

withdraw transaction	refresh transaction
1. $\alpha \leftarrow \mathsf{DAP.ExtractPour}(\mathsf{pp}, \mathsf{td}_{\mathsf{DAP}}, \mathsf{tx}_{\mathsf{wd}})$. 2. Parse α as $(\mathbf{d}, \mathsf{ask}_{\mathbf{d}}, v, \mathsf{apk}, \mathsf{pub}, \mathsf{sec}, v_{\mathsf{pub}}, \mathsf{info}, \Delta, \mathbf{c})$. 3. Retrieve $\mathbb{d} := \mathsf{SimNotes}[\mathbf{d}]$. 4. Invoke $\mathsf{T_p.RefreshTicket}(\mathbb{d}, \mathsf{apk}, v_{\mathsf{pub}})$. 5. **Receive from $\mathsf{T_p}$:** $(\mathtt{withdraw}, \mathbb{c}, v_{\mathsf{pub}})$. 6. Set $\mathsf{SimNotes}[\mathbf{c}] := \mathbb{c}$.	1. $\alpha \leftarrow \mathsf{DAP.ExtractPour}(\mathsf{pp}, \mathsf{td}_{\mathsf{DAP}}, \mathsf{tx}_{\mathsf{ref}})$. 2. Parse α as $(\mathbf{t}, \mathsf{ask}_{\mathbf{t}}, v, \mathsf{apk}, \mathsf{pub}, \mathsf{sec}, v_{\mathsf{pub}}, \mathsf{info}, \Delta, \mathbf{t}')$. 3. Retrieve $\mathbb{t} := \mathsf{SimNotes}[\mathbf{t}]$. 4. Invoke $\mathsf{T_p.RefreshTicket}(\mathbb{t}, \mathsf{apk}, v_{\mathsf{pub}})$. 5. **Receive from $\mathsf{T_p}$:** $(\mathtt{refresh}, \mathbb{t}', v_{\mathsf{pub}})$. 6. Set $\mathsf{SimNotes}[\mathbf{t}'] := \mathbb{t}'$.

Adversarial probabilistic payments. When $\mathcal{A}$ (in the role of the Sender) engages in the DAM.PourTicket protocol with an honest receiving party $\mathcal{P}_{\mathsf{r}}$, $\mathcal{S}$ engages in the PourTicket protocol with $\mathcal{A}$ in the role of the Receiver, and calls $\mathsf{T_p.PourTicket}(\mathcal{P}_{\mathsf{r}})$. Then, $\mathcal{A}$ performs the following actions.

1. **Receive from $\mathsf{T_p}$:** $\mathcal{P}_{\mathsf{r}}$'s $(\mathsf{sid}, \mathcal{D}, v_{\mathsf{pub}})$.
2. Compute $(\mathsf{pk}_{\mathsf{FMT}}, \mathsf{sk}_{\mathsf{FMT}}) \leftarrow \mathsf{FMT.SimSetup}(\mathsf{pp}_{\mathsf{FMT}}, \mathsf{td}_{\mathsf{FMT}})$.
3. Compute $(\mathsf{pk}_{\mathsf{SIG}}, \mathsf{sk}_{\mathsf{SIG}}) \leftarrow \mathsf{SIG.Keygen}(\mathsf{pp}_{\mathsf{SIG}})$.
4. Compute $\gamma_{\mathcal{D}} \leftarrow \mathsf{DB.Comm}(\mathcal{D})$.
5. Construct $\mathsf{spk} := (\mathsf{pk}_{\mathsf{FMT}}, \mathsf{pk}_{\mathsf{SIG}})$.
6. **Send to $\mathcal{A}$:** $(\mathsf{sid}, \mathsf{spk}, \mathcal{D}, \gamma_{\mathcal{D}}, v_{\mathsf{pub}})$.
7. **Receive from $\mathcal{A}$:** $(m_0, \omega_0, \omega_1, r_0, \pi_{\mathsf{pt}})$.
8. Parse m_0 as $(\mathsf{sid}', \mathsf{spk}', v'_{\mathsf{pub}}, \gamma'_{\mathcal{D}}, c_{\mathsf{FMT}}, \mathsf{arlt})$.
9. $\mathbb{x} := (\mathbf{L}.\mathsf{Len}, \mathbf{L}.\mathsf{SNRoot}, \mathbf{L}.\mathsf{CMRoot}, \mathsf{pk}_{\mathsf{SIG}}, \omega_0, \omega_1)$.
10. If $\mathsf{COMM}_{r_0}(m_0) \neq \omega_0$: output `fail`.
11. If $(\mathsf{sid}', \mathsf{spk}', v'_{\mathsf{pub}}, \gamma'_{\mathcal{D}}) \neq (\mathsf{sid}, \mathsf{spk}, v_{\mathsf{pub}}, \gamma_{\mathcal{D}})$: output `fail`.
12. Check that $\mathsf{NIZK.Verify}(\mathsf{crs}_{\mathsf{pt}}, \mathbb{x}, \pi_{\mathsf{pt}}) = 1$.
13. Compute $\mathbb{w} \leftarrow \mathsf{NIZK.Extract}(\mathsf{crs}_{\mathsf{pt}}, \mathsf{td}_{\mathsf{pt}}, \pi_{\mathsf{pt}})$.
14. Parse $\mathbb{w}$ as $\begin{pmatrix} \mathbf{t}, \mathsf{ask}_{\mathbf{t}}, \mathbf{d}, \mathsf{ask}_{\mathbf{d}} & \mathbf{c}, \mathsf{tx}_{\mathsf{p}}, \mathsf{info} & \pi_{\mathcal{R}}, \pi_{\mathsf{sn}}, \pi_{\mathsf{cm}}, \pi_{\mathcal{D}} & c_{\mathsf{FMT}}, r_0, r_1 \\ \mathsf{sid}, \mathsf{spk}, v_{\mathsf{pub}} & \mathsf{dst}, \mathsf{arlt}, \mathsf{wrlt} & \mathsf{actr}, \mathsf{wctr}, \gamma_{\mathcal{D}} & \end{pmatrix}$.
15. Retrieve $\mathbb{t} := \mathsf{SimNotes}[\mathbf{t}]$.
16. Set $\mathtt{ok}? := \mathbb{t}$.
17. **Send to $\mathsf{T_p}$:** $\mathtt{ok}?$.
18. **Receive from $\mathsf{T_p}$:** `final`.
19. If $\mathtt{final} = \mathtt{abort}$: **Send to $\mathcal{A}$:** `abort`.
20. If $\mathtt{final} \in \{0, 1\}$:
    (a) $\mathsf{sk}'_{\mathsf{FMT}} \leftarrow \mathsf{FMT.SimDecrypt}(\mathsf{pp}_{\mathsf{FMT}}, \mathsf{td}_{\mathsf{FMT}}, \mathsf{sk}_{\mathsf{FMT}}, \mathtt{ok}?)$.
    (b) If $\mathtt{final} = 0$: set $m' = \varnothing$.
    (c) If $\mathtt{final} = 1$: set $m' = \mathsf{FMT.ExtDecrypt}(\mathsf{pp}_{\mathsf{FMT}}, \mathsf{td}_{\mathsf{FMT}}, c_{\mathsf{FMT}})$.
    (d) **Send to $\mathcal{A}$:** $(m', \mathsf{sk}_{\mathsf{FMT}})$.

Adversarial pour-ticket transactions. When $\mathcal{A}$ posts a pour-ticket transaction $\mathsf{tx}_{\mathsf{pt}}$ to $\mathbf{L}$, $\mathcal{S}$ performs the following actions.

1. Retrieve $(\mathsf{tok}_{\mathtt{tk}}, \mathsf{sid}, v_{\mathsf{pub}}) := \mathsf{SimPourTk}[\mathsf{tx}_{\mathsf{pt}}.\mathsf{sn}]$.
2. Invoke $\mathsf{T_p.RedeemTicket}(\mathsf{tok}_{\mathtt{tk}}, \mathsf{sid}, v_{\mathsf{pub}})$.
3. Discard any messages received as a result of this invocation.

C.1.3 Trusted party messages:

When $\mathcal{S}$ receives messages from the trusted party $\mathsf{T_p}$, it performs the following actions.

Mints. When $\mathsf{T_p}$ sends a mint message $(\mathtt{mint}, \mathbb{n}, v)$ to $\mathcal{S}$, where $\mathtt{mint} \in \{\mathtt{mintcn}, \mathtt{minttk}, \mathtt{mintdp}\}$, $\mathcal{S}$ performs the following actions.

if $\mathtt{mint} = \mathtt{mintcn}$	if $\mathtt{mint} = \mathtt{mintdp}$	if $\mathtt{mint} = \mathtt{minttk}$
1. Set $\mathsf{pub} := \mathtt{cn}$.	1. Set $\mathsf{pub} := \mathtt{dp}$.	1. Set $\mathsf{pub} := \mathtt{tk}$.
2. $(\mathbf{n}, \mathsf{tx_m}) \leftarrow \mathsf{DAP.SimMint}^{\mathbf{L}}(\mathsf{pp_{DAP}}, \mathsf{td_{DAP}}, v, \mathsf{pub})$. 3. Set $\mathsf{SimNotes}[\mathbf{n}] := \mathbb{n}$. 4. Append $\mathsf{tx_m}$ to $\mathbf{L}$.		

Pour-coins. When $\mathsf{T_p}$ sends a pour-coin message for a pour-coin transaction $(\mathtt{pourcn}, m, [\mathbb{n}_j]_1^n, v_{\mathsf{pub}})$ to all parties, $\mathcal{S}$ performs the following actions.

- If no additional messages are received:
  1. Compute $([\mathbf{n}_j]_1^n, \mathsf{tx_p}) \leftarrow \mathsf{DAP.SimPour}^{\mathbf{L}}(\mathsf{pp_{DAP}}, \mathsf{td_{DAP}}, m, n, v_{\mathsf{pub}}, 0, \perp)$.
  2. For each $j = 1$ to n: set $\mathsf{SimNotes}[\mathbf{n}_j] := \mathbb{n}_j$.
  3. Append $\mathsf{tx_p}$ to $\mathbf{L}$.
- If $p \leq n$ additional messages of the following forms are received:

$[(\mathtt{prvpc}, \mathtt{cn}, \mathbb{c}_k, v_k, \mathsf{a}_k)]_1^p$	$[(\mathtt{prvpc}, \mathtt{dp}, \mathbb{d}_k, v_k, \mathsf{a}_k, \mathcal{R}_k)]_1^p$	$[(\mathtt{prvpc}, \mathtt{tk}, \mathbb{t}_k, v_k, \mathsf{a}_k, \mathsf{info}_k^{\mathbb{d}})]_1^p$
1. For $k = 1$ to p: (a) $(\mathbf{n}_k, \mathsf{tx_m}) \leftarrow \mathsf{DAM.MintCoin}(\mathsf{pp}, v_k, \mathsf{a}_k)$.	1. For $k = 1$ to p: (a) $\alpha_{\mathcal{R}_k} \leftarrow \mathsf{RST.Comm}(\mathcal{R}_k)$. (b) $(\mathbf{n}_k, \mathsf{tx_m}) \leftarrow \mathsf{DAM.MintDeposit}(\mathsf{pp}, v_k, \mathsf{a}_k, \alpha_{\mathcal{R}_k})$.	1. For $k = 1$ to p: (a) retrieve $\mathbf{d}_k$ s.t. $\mathsf{SimNotes}[\mathbf{d}_k] = \mathsf{info}_k^{\mathbb{d}}.\mathbb{d}$. (b) $(\mathbf{n}_k, \mathsf{tx_m}) \leftarrow \mathsf{DAM.MintTicket}(\mathsf{pp}, v_k, \mathsf{a}_k, \mathbf{d}_k)$.
3. $([\mathbf{n}_j]_1^n, \mathsf{tx_p}) \leftarrow \mathsf{DAP.SimPour}(\mathsf{pp_{DAP}}, \mathsf{td_{DAP}}, m, n, v_{\mathsf{pub}}, 0, [\mathbf{d}_k]_1^p)$. 4. For each $j = 1$ to n: set $\mathsf{SimNotes}[\mathbf{n}_j] := \mathbb{n}_j$. 5. Append $\mathsf{tx_p}$ to $\mathbf{L}$.		

Deposit withdrawals and ticket refreshes. When $\mathsf{T_p}$ sends a withdraw message $(\mathtt{withdraw}, \mathbb{c}, v_{\mathsf{pub}})$ or a refresh message $(\mathtt{refresh}, \mathbb{t}', v_{\mathsf{pub}})$ to $\mathcal{S}$, $\mathcal{S}$ performs the following actions.

- If no additional messages are received:

withdraw transaction	refresh transaction
1. $(\mathbf{c}, \mathsf{tx_p}) \leftarrow \mathsf{DAP.SimPour}(\mathsf{pp_{DAP}}, \mathsf{td_{DAP}}, 1, 1, v_{\mathsf{pub}}, \Delta_{\mathsf{w}}, \perp)$. 2. Set $\mathsf{SimNotes}[\mathbf{c}] := \mathbb{c}$. 3. Append $\mathsf{tx_p}$ to $\mathbf{L}$.	1. $(\mathbf{t}', \mathsf{tx_p}) \leftarrow \mathsf{DAP.SimPour}(\mathsf{pp_{DAP}}, \mathsf{td_{DAP}}, 1, 1, v_{\mathsf{pub}}, \Delta_{\mathsf{r}}, \perp)$. 2. Set $\mathsf{SimNotes}[\mathbf{t}'] := \mathbb{t}'$. 3. Append $\mathsf{tx_p}$ to $\mathbf{L}$.

- If additional messages of the following forms are received:

withdraw transaction $(\mathtt{withdraw}, \mathtt{cn}, \mathbb{c}, v, \mathsf{a})$	refresh transaction $(\mathtt{refresh}, \mathtt{tk}, \mathbb{t}', v, \mathsf{a}, \mathsf{info}^{\mathbb{d}})$
1. $(\mathbf{c}, \mathsf{tx_m}) \leftarrow \mathsf{DAM.MintCoin}(\mathsf{pp_{DAP}}, v, \mathsf{a})$. 2. $(\mathbf{c}, \mathsf{tx_p}) \leftarrow \mathsf{DAP.SimPour}(\mathsf{pp_{DAP}}, \mathsf{td_{DAP}}, 1, 1, v_{\mathsf{pub}}, \Delta_{\mathsf{w}}, \mathbf{c})$. 3. Set $\mathsf{SimNotes}[\mathbf{c}] := \mathbb{c}$. 4. Append $\mathsf{tx_p}$ to $\mathbf{L}$.	1. Retrieve $\mathbf{d}$ such that $\mathsf{SimNotes}[\mathbf{d}] = \mathsf{info}^{\mathbb{d}}.\mathbb{d}$. 2. $(\mathbf{t}', \mathsf{tx_m}) \leftarrow \mathsf{DAM.MintTicket}(\mathsf{pp_{DAP}}, v, \mathsf{a}, \mathbf{d})$. 3. $(\mathbf{t}', \mathsf{tx_p}) \leftarrow \mathsf{DAP.SimPour}(\mathsf{pp_{DAP}}, \mathsf{td_{DAP}}, 1, 1, v_{\mathsf{pub}}, \Delta_{\mathsf{r}}, \mathbf{t}')$. 4. Set $\mathsf{SimNotes}[\mathbf{t}'] := \mathbb{t}'$. 5. Append $\mathsf{tx_p}$ to $\mathbf{L}$.

Probabilistic payments. When an honest party $\mathcal{P}_{\mathsf{s}}$, operating as the sender, initiates the pour-ticket protocol PourTicket via $\mathsf{T_p}$ with receiving party $\mathcal{S}$, $\mathcal{S}$, acting in the role of the sender, engages in the DAM.PourTicket protocol with $\mathcal{A}$ as follows.

1. **Receive from $\mathsf{T_p}$:** $\mathtt{init}$.
2. Initiate the DAM.PourTicket protocol with $\mathcal{A}$.
3. **Receive from $\mathcal{A}$:** $(\mathsf{sid}, \mathsf{spk}, \mathcal{D}, \gamma_{\mathcal{D}}, v_{\mathsf{pub}})$.
4. Parse sid as $(\mathsf{apk_c}, \mathsf{tw_r})$.
5. Parse $\mathsf{apk_c}.\mathsf{spec}$ as $(\mathfrak{a}_{\mathsf{r}}, \mathfrak{w}_{\mathsf{r}}, p_{\mathsf{r}}, V_{\mathsf{r}})$.
6. **Send to $\mathsf{T_p}$:** $(\mathsf{sid}, \mathcal{D}, v_{\mathsf{pub}})$.

7. **Receive from $\mathsf{T_p}$:** $(\mathtt{pourtk}, \mathsf{avg}, \mathsf{worst}, \mathsf{result})$.
8. If $\mathsf{avg} = 0$:
   (a) sample random average-case rate limit tag arlt.
   (b) set $\mathsf{SimARates}[\mathsf{sid}] := \mathsf{SimARates}[\mathsf{sid}] \cup \{\mathsf{arlt}\}$.
9. If $\mathsf{avg} = 1$: sample a random average-case rate limit tag arlt from $\mathsf{SimARates}[\mathsf{sid}]$.
10. If $\mathsf{result} = \mathtt{null}$:
    (a) set $b := 0$.
    (b) sample random double spend tag dst.
11. If $\mathsf{result} = (\mathtt{macro}, \mathsf{tok}_{\mathtt{tk}})$:
    (a) set $b := 1$.
    (b) compute $t := \mathsf{CRH}(\mathsf{pk}_{\mathsf{SIG}})$.
    (c) sample random double spend tag dst.
    (d) set $\mathsf{SimTokens}[\mathsf{tok}_{\mathtt{tk}}] := (t, \mathsf{dst})$.
12. If $\mathsf{result} = (\mathtt{macro}, \mathsf{tok}_{\mathtt{tk}}, \mathsf{tok}_{\mathtt{dp}})$:
    (a) set $b := 1$.
    (b) retrieve $(t', \mathsf{dst}') := \mathsf{SimTokens}[\mathsf{tok}_{\mathtt{tk}}]$.
    (c) if $\mathsf{SimTokens}[\mathsf{tok}_{\mathtt{dp}}] = \perp$:
        i. sample sn and id randomly.
        ii. set $\mathsf{SimTokens}[\mathsf{tok}_{\mathtt{dp}}] = (\mathsf{sn}, \mathsf{id})$.
    (d) else: retrieve $(\mathsf{sn}, \mathsf{id}) := \mathsf{SimTokens}[\mathsf{tok}_{\mathtt{dp}}]$.
    (e) compute $t' := \mathsf{CRH}(\mathsf{pk}_{\mathsf{SIG}})$.
    (f) set $s := (\mathsf{sn}, \mathsf{id})$
    (g) Construct line $\ell(X) := (\mathsf{dst}' - s)/(t' - 0) + s$.
    (h) compute $\mathsf{dst} := \ell(t)$.
13. If $\mathsf{result} \neq \mathtt{null}$:
    (a) if $\mathsf{worst} = 0$:
        i. sample random worst-case rate limit tag wrlt.
        ii. set $\mathsf{SimWRates}[\mathsf{sid}] := \mathsf{SimWRates}[\mathsf{sid}] \cup \{\mathsf{wrlt}\}$.
    (b) if $\mathsf{worst} = 1$: sample a random worst-case rate limit tag wrlt from $\mathsf{SimWRates}[\mathsf{sid}]$.
14. $(\mathbf{c}, \mathsf{tx_m}) \leftarrow \mathsf{DAM.MintCoin}(\mathsf{pp_{DAP}}, V_{\mathsf{r}}, \mathsf{apk}_{\mathbf{c}})$.
15. $(\mathbf{c}, \mathsf{tx_p}) \leftarrow \mathsf{DAP.SimPour}(\mathsf{pp_{DAP}}, \mathsf{td_{DAP}}, 1, 1, v_{\mathsf{pub}}, 0, \mathbf{c})$.
16. Set $\mathsf{SimPourTk}[\mathsf{tx_p.sn}] := (\mathsf{tok}_{\mathtt{tk}}, \mathsf{sid}, v_{\mathsf{pub}})$.
17. Sample commitment randomness r_0, r_1.
18. Set $m_1 := (\mathsf{tx_p}, \mathsf{dst}, \mathsf{wrlt})$.
19. $c_{\mathsf{FMT}} \leftarrow \mathsf{FMT.SimEncrypt}(\mathsf{pp_{FMT}}, \mathsf{td_{FMT}}, \mathsf{spk.pk_{FMT}}, b, m_1 \| r_1)$.
20. Set $\omega_1 := \mathsf{COMM}_{r_1}(\mathsf{tx_p}, \mathsf{dst}, \mathsf{wrlt})$.
21. Set $\omega_0 := \mathsf{COMM}_{r_0}(\mathsf{sid}, \mathsf{spk}, v_{\mathsf{pub}}, \gamma_{\mathcal{D}}, c_{\mathsf{FMT}}, \mathsf{arlt})$.
22. Set $\mathbb{x} := (\mathbf{L}.\mathsf{Len}, \mathbf{L}.\mathsf{SNRoot}, \mathbf{L}.\mathsf{CMRoot}, \mathsf{pk_{SIG}}, \omega_0, \omega_1)$.
23. $\pi_{\mathsf{pt}} \leftarrow \mathsf{NIZK.Simulate}(\mathsf{crs_{pt}}, \mathsf{td_{pt}}, \mathbb{x})$.
24. **Send to $\mathcal{A}$:** $(m_0, \omega_0, \omega_1, r_0, \pi_{\mathsf{pt}})$.
25. **Receive from $\mathcal{A}$:** $(m', \mathsf{sk_{FMT}})$.
26. If $\mathsf{FMT.Decrypt}(\mathsf{pp_{FMT}}, \mathsf{sk_{FMT}}, c_{\mathsf{FMT}}) = m'$: **Send to $\mathsf{T_p}$:** $\mathtt{ok}? := 1$.
27. If $\mathsf{FMT.Decrypt}(\mathsf{pp_{FMT}}, \mathsf{sk_{FMT}}, c_{\mathsf{FMT}}) \neq m'$: **Send to $\mathsf{T_p}$:** $\mathtt{ok}? := 0$.

Pour-ticket notification. When $\mathsf{T_p}$ sends a pour-ticket message $(\mathtt{pourtk}, \mathbb{c}, v_{\mathsf{pub}})$ *not in response to an invocation by* $\mathcal{S}$, $\mathcal{S}$ performs the following actions:

1. If $\mathcal{S}$ also receives a message of the form $(\mathtt{ptk}, \mathtt{cn}, \mathbb{c}, v, \mathsf{a_r})$:
   (a) compute $(\mathbf{c}, \mathsf{tx_m}) \leftarrow \mathsf{DAM.MintCoin}(\mathsf{pp_{DAP}}, V_{\mathsf{r}}, \mathsf{a_r})$.
   (b) $(\mathbf{c}, \mathsf{tx_p}) \leftarrow \mathsf{DAP.SimPour}(\mathsf{pp_{DAP}}, \mathsf{td_{DAP}}, 1, 1, v_{\mathsf{pub}}, 0, \mathbf{c})$.

2. If no such message is received: compute $(\mathbf{c}, \mathsf{tx_p}) \leftarrow \mathsf{DAP.SimPour}(\mathsf{pp_{DAP}}, \mathsf{td_{DAP}}, 1, 1, v_{\mathsf{pub}}, 0, \perp)$.
3. Parse $\mathsf{tx_p}$ as $(\mathsf{ts}, 0, \mathsf{sn}, \mathsf{cm}, v_{\mathsf{pub}}, \mathsf{info}, *)$.
4. Sample a one-time signature key pair $(\mathsf{pk_{SIG}}, \mathsf{sk_{SIG}}) \leftarrow \mathsf{SIG.Keygen}(\mathsf{pp.pp_{SIG}})$.
5. Sample random double spend tag dst, commitment ω_0, randomness r_1, and worst-case rate limit tag wrlt.
6. Set $\omega_1 := \mathsf{COMM}_{r_1}(\mathsf{tx_p}, \mathsf{dst}, \mathsf{wrlt})$.
7. Simulate $\pi_{\mathsf{pt}} \leftarrow \mathsf{NIZK.Simulate}(\mathsf{crs_{pt}}, \mathsf{td_{pt}}, (\mathbf{L}.\mathsf{Len}, \mathbf{L}.\mathsf{SNRoot}, \mathbf{L}.\mathsf{CMRoot}, \mathsf{pk_{SIG}}, \omega_0, \omega_1))$.
8. $m_{\mathsf{SIG}} := (\mathsf{tx_p}, \mathsf{dst}, \omega_0, \omega_1, r_1, \pi_{\mathsf{pt}})$.
9. $\sigma \leftarrow \mathsf{SIG.Sign}(\mathsf{pp_{SIG}}, \mathsf{sk_{SIG}}, m_{\mathsf{SIG}})$.
10. $*' := (*, (\mathsf{pk_{SIG}}, m_{\mathsf{SIG}}, \sigma))$.
11. $\mathsf{tx_{pt}} := \mathsf{tx'_p}$ where $\mathsf{tx'_p} := (\mathsf{ts}, 0, \mathsf{sn}, \mathsf{cm}, v_{\mathsf{pub}}, \mathsf{info}, *')$.
12. Push $\mathsf{tx_{pt}}$ to $\mathbf{L}$.

Double spend notification. When $\mathsf{T_p}$ sends a double-spend message $(\texttt{doublespend}, \mathsf{tok_{dp}})$ *without being invoked by* $\mathcal{S}$, $\mathcal{S}$ performs the following actions:

1. If $\mathsf{SimTokens}[\mathsf{tok_{dp}}] = \perp$:
   (a) sample a random serial number sn and a random deposit identifier id.
   (b) set $\mathsf{SimTokens}[\mathsf{tok_{dp}}] := (\mathsf{sn}, \mathsf{id})$.
2. If $\mathsf{SimTokens}[\mathsf{tok_{dp}}] \neq \perp$: retrieve $(\mathsf{sn}, \mathsf{id}) := \mathsf{SimTokens}[\mathsf{tok_{dp}}]$.
3. Construct two simulated pour-ticket transactions $\mathsf{tx_{pt}}$ and $\mathsf{tx'_{pt}}$ that are valid and consistent with sn and id.
4. Set $\mathsf{tx_{pun}} := (\mathsf{tx_{pt}}, \mathsf{tx'_{pt}}, \mathsf{sn}, \mathsf{id})$
5. Post $\mathsf{tx_{pun}}$ to $\mathbf{L}$.

All transactions. In addition to the specific transaction handling above, $\mathcal{S}$ monitors the output of $\mathcal{A}$ (including the output of the NIZK and DAP extraction algorithms), and sends $\mathcal{A}$ abort if any of the following cases occur:

1. $\mathcal{A}$ outputs two commitment openings for a single commitment in the scheme COMM.
2. $\mathcal{A}$ outputs two different pre-images for the same output of the hash function CRH or PRF.

C.2 Proof of security by hybrid argument

We now show that the output distribution of the ideal world experiment is computationally indistinguishable from the output distribution of the real world experiment. We demonstrate this via a series of hybrids defined by games $\mathcal{G}_i$. Denote by $\mathsf{Output}(\mathcal{G}_i(\mathcal{A}))$ the output of $\mathcal{A}$ at the end of game $\mathcal{G}_i$.

$\mathcal{G}_0$: This game corresponds to the real experiment.

$\mathcal{G}_1$: In this game, the trusted parameter generator modifies the DAM public parameters by replacing the contained honestly-generated pour-ticket NIZK CRS with a simulated one (that is, one generated via invoking NIZK.SimSetup). Additionally, each pour-ticket NIZK π_{pt} sent from an honest party to $\mathcal{A}$ is replaced with a simulated proof, and the knowledge extractor is run on each π_{pt} produced by $\mathcal{A}$. If any extraction fails, the experiment aborts and outputs `abort`. Assuming that the NIZK scheme NIZK is simulation-extractable and zero knowledge, $\mathsf{Output}(\mathcal{G}_0(\mathcal{A}))$ is computationally indistinguishable from $\mathsf{Output}(\mathcal{G}_1(\mathcal{A}))$.

$\mathcal{G}_2$: In this game, the trusted parameter generator modifies the DAM public parameters by replacing the contained honestly-generated DAP parameters with simulated ones (that is, ones generated by invoking DAP.SimSetup). Additionally, each DAP Mint and Pour generated by an uncorrupted party is replaced with the corresponding simulated transaction created via DAP.SimMint and DAP.SimPour respectively. Similarly DAP.ExtractMint and DAP.ExtractPour are run whenever $\mathcal{A}$ outputs a valid mint or pour transaction. If any extraction fails, the experiment abourts and outputs `abort`. Assuming that the DAP scheme is secure in the sense of Definition B.2, then by Lemma C.1, we have that $\mathsf{Output}(\mathcal{G}_1(\mathcal{A}))$ is computationally indistinguishable from $\mathsf{Output}(\mathcal{G}_2(\mathcal{A}))$.

$\mathcal{G}_3$: In this game, the trusted parameter generator modifies the DAM public parameters by replacing the contained honestly generated FMT parameters with simulated ones. By Claim A.5, we have that $\mathsf{Output}\,(\mathcal{G}_2(\mathcal{A}))$ is computationally indistinguishable from $\mathsf{Output}\,(\mathcal{G}_3(\mathcal{A}))$.

$\mathcal{G}_4$: Whenever an honest party acting as Sender engages in pour-ticket protocol $\mathsf{DAM.PourTicket}^{\mathbf{L}}$ with the $\mathcal{A}$ acting as a Receiver, the honest party samples a bit b such that $b = 1$ with probability p, and simulates running the protocol. It runs $\mathsf{FMT.SimEncrypt}$ using the trapdoor $\mathsf{td}_{\mathsf{FMT}}$ to simulate the construction of the FMT ciphertext c_{FMT} such that the ciphertext will decrypt correctly with probability 1 when $b = 1$, and when $b = 0$ will decrypt to $\varnothing$. By Claim A.7 and Lemma C.2, we have that $\mathsf{Output}\,(\mathcal{G}_3(\mathcal{A}))$ is computationally indistinguishable from $\mathsf{Output}\,(\mathcal{G}_4(\mathcal{A}))$.

$\mathcal{G}_5$: Whenever the real-world adversary $\mathcal{A}$ engages in $\mathsf{DAM.PourTicket}^{\mathbf{L}}$ acting as a Sender to receive payment from an honest party $\mathcal{P}_{\mathsf{s}}$, the honest party first runs $\mathsf{FMT.SimKeygen}$ on the trapdoor $\mathsf{td}_{\mathsf{FMT}}$ and sends the resulting $\mathsf{pk}_{\mathsf{FMT}}$ to $\mathcal{A}$ as part of spk in the first move of the protocol. Next, upon receiving c from $\mathcal{A}$ in the second move of the protocol, the honest party runs $(m, r) \leftarrow \mathsf{FMT.ExtDecrypt}(\mathsf{pp}_{\mathsf{FMT}}, \mathsf{td}_{\mathsf{FMT}}, \mathsf{sk}_{\mathsf{FMT}}, c)$ with probability 1. If the decryption of c is not consistent with the witness extracted from π_{pt}, abort and output $\mathtt{abort}$ to $\mathcal{A}$. By Claim A.7 and Lemma C.3, we have that $\mathsf{Output}\,(\mathcal{G}_4(\mathcal{A}))$ is computationally indistinguishable from $\mathsf{Output}\,(\mathcal{G}_5(\mathcal{A}))$.

$\mathcal{G}_6$: Whenever the real-world adversary $\mathcal{A}$ posts a *valid* pour-ticket transaction $\mathsf{tx}_{\mathsf{pt}}$ containing a signature σ under $\mathsf{pk}_{\mathsf{SIG}}$ such that $\mathsf{pk}_{\mathsf{SIG}}$ was generated by an honest party, but the tuple $(m_{\mathsf{SIG}}, \sigma)$ was not a message/signature pair generated by the honest party, abort and output $\mathtt{abort}$. Under the assumption that the signature scheme is strongly unforgeable, then by Lemma C.4 we have that $\mathsf{Output}\,(\mathcal{G}_5(\mathcal{A}))$ is computationally indistinguishable from $\mathsf{Output}\,(\mathcal{G}_6(\mathcal{A}))$.

$\mathcal{G}_7$: Whenever the real-world adversary $\mathcal{A}$ outputs two distinct openings to the same commitment for the scheme COMM, abort and output $\mathtt{abort}$. Note that if COMM is a computationally binding commitment scheme, this occurs with at most negligible probability. Thus we have that $\mathsf{Output}\,(\mathcal{G}_6(\mathcal{A}))$ is computationally indistinguishable from $\mathsf{Output}\,(\mathcal{G}_7(\mathcal{A}))$.

$\mathcal{G}_8$: Whenever the real-world adversary $\mathcal{A}$ outputs two distinct pre-images to the same output of the function CRH or PRF, abort and output $\mathtt{abort}$. Note that if these functions are collision resistant, this will occur with at most negligible probability. Thus we have that $\mathsf{Output}\,(\mathcal{G}_7(\mathcal{A}))$ is computationally indistinguishable from $\mathsf{Output}\,(\mathcal{G}_8(\mathcal{A}))$.

$\mathcal{G}_9$: Whenever the real-world adversary $\mathcal{A}$ receives a ticket that has not been spent from an honest party, replace the slope of the double spend tag with a random slope drawn from the appropriate domain. If PRF is a PRF, this happens with at most negligible probability. Thus we have that $\mathsf{Output}\,(\mathcal{G}_8(\mathcal{A}))$ is computationally indistinguishable from $\mathsf{Output}\,(\mathcal{G}_9(\mathcal{A}))$.

$\mathcal{G}_{10}$: Whenever the real-world adversary $\mathcal{A}$ initiates a $\mathsf{PourTicket}$ interaction on an (extracted) deposit $\mathbf{d}$, and the total of the payment rates made by $\mathcal{A}$ on $\mathbf{d}$ exceeds the value $\mathfrak{s}$ that is embedded in $\mathbf{d}.\mathsf{sec}$, abort the simulation and output the distinguished error message rates. If CRH is collision resistant, this happens with at most negligible probability. Thus, by Lemma C.5, $\mathsf{Output}\,(\mathcal{G}_9(\mathcal{A}))$ is computationally indistinguishable from $\mathsf{Output}\,(\mathcal{G}_{10}(\mathcal{A}))$.

We note that the final hybrid is distributed identically to the operation of $\mathcal{S}$ from the point of view of $\mathcal{A}$. By summation over the previous hybrids we show that $\mathcal{A}$'s advantage in distinguishing the interaction with $\mathcal{S}$ from the interaction with the real protocol is at most negligible in λ. We now complete the proof via the following lemmas.

Lemma C.1. *No efficient adversary $\mathcal{A}$ can distinguish between the distributions $\mathcal{G}_1$ and $\mathcal{G}_2$.*

Proof. The proof relies on the security of the underlying DAP scheme, as defined in Appendix B. Recall that $\mathcal{G}_1$ uses the real DAP scheme DAP, and $\mathcal{G}_2$ uses the DAP simulation routines to construct DAP transactions. Let $\mathcal{A}$ be a real-world adversary that succeeds in distinguishing $\mathcal{G}_1$ from $\mathcal{G}_2$ with non-negligible advantage. We show how to construct a new adversary $\mathcal{A}'$ that succeeds with non-negligible advantage against the DAP scheme according to Definition B.2. $\mathcal{A}'$ runs the experiment of Definition B.2, and simultaneously interacts with $\mathcal{A}$ which runs the DAM protocol as follows.

First, $\mathcal{A}'$ receives $\mathsf{pp}_{\mathsf{DAP}}$ from the DAP experiment. It then embeds these parameters in pp and sends them to $\mathcal{A}$. Now $\mathcal{A}'$ runs the normal DAM protocol with $\mathcal{A}$, taking the role of all honest parties. However each time $\mathcal{A}'$ is required to 1. create a new address for an honest party, 2. construct a DAP Mint transaction from an honest party, or 3. create a

Pour transaction from an honest party, $\mathcal{A}'$ queries the DAP experiment to obtain the necessary values. It then uses the result (address or transaction) in place of the normal value that would be used in the DAM protocol. If the experiment aborts prematurely, abort and return `abort` to $\mathcal{A}$.

We note that if $\mathcal{A}'$ is engaged in the REAL experiment for the DAP scheme, then the distribution of the values received by $\mathcal{A}$ is identical to that of $\mathcal{G}_1$. On the other hand, when $\mathcal{A}'$ is engaged in the IDEALexperiment for the DAP scheme, then the distribution of values received by $\mathcal{A}$ is identical to $\mathcal{G}_2$. Thus, if $\mathcal{A}$ successfully distinguishes $\mathcal{G}_1$ from $\mathcal{G}_2$ with advantage that is non-negligible in λ, then we obtain a distinguisher that succeeds against the DAP scheme. This contradicts our assumption that the DAP scheme is secure. □

Lemma C.2. *No efficient adversary $\mathcal{A}$ can distinguish between the distributions $\mathcal{G}_3$ and $\mathcal{G}_4$.*

Proof. The proof relies on fractional hiding property of the underlying FMT scheme. Let $\mathcal{A}$ be an adversary that distinguishes $\mathcal{G}_3$ from $\mathcal{G}_4$ with non-negligible advantage. Then we construct a new adversary $\mathcal{A}'$ that succeeds in distinguishing the two distributions defined in Claim A.7 with non-negligible advantage. $\mathcal{A}'$ selects one PourTicket interaction in the first such transaction, and emulates the honest sender party. $\mathcal{A}'$ receives $\mathsf{pk}_{\mathsf{FMT}}$ from $\mathcal{A}$, and selects the appropriate pour transaction tx_{p} and double spend tag dst to use as the message m. $\mathcal{A}'$ forwards this $\mathsf{pk}_{\mathsf{FMT}}$ and m to the challenger of the FMT fractional hiding game. It receives a ciphertext c in return, and passes it to $\mathcal{A}$. $\mathcal{A}'$ then outputs whatever $\mathcal{A}$ outputs. When the challenger encrypts the ciphertext using FMT.Encrypt, the view of $\mathcal{A}$ is identical to $\mathcal{G}_3$. When the challenger encrypts using FMT.SimEncrypt, the view of $\mathcal{A}$ is identical to that of $\mathcal{G}_4$.

We repeat this process via a hybrid argument, systematically modifying each PourTicket interaction as above until all such interactions have been modified. The resulting distribution is identical to $\mathcal{G}_4$. By definition if $\mathcal{A}$'s output can be distinguished between $\mathcal{G}_3$ and $\mathcal{G}_4$ then there must exist one pair of intermediate hybrids in which $\mathcal{A}'$ succeeds in distinguishing the distributions with non-negligible probability. This implies that $\mathcal{A}'$ can distinguish the two distributions of Claim A.7 with non-negligible advantage, which contradicts the assumption. □

Lemma C.3. *No efficient adversary $\mathcal{A}$ can distinguish between the distributions $\mathcal{G}_4$ and $\mathcal{G}_5$.*

Proof. The proof relies on the fractional binding property of the underlying FMT scheme. Let $\mathcal{A}$ be an adversary that distinguishes between the outputs of the two games with non-negligible advantage. Then we construct a new adversary $\mathcal{A}'$ that succeeds in distinguishing the two distributions defined in Claim A.8 with non-negligible advantage. $\mathcal{A}'$ selects one PourTicket interaction in the first such transaction, and replaces the honestly-generated public key with embeds a simulated public key in place of the corresponding values in the interaction. Next, when $\mathcal{A}$ outputs c, $\mathcal{A}'$ runs FMT.ExtDecrypt to extract the underlying message m and simulates a secret key $\mathsf{sk}_{\mathsf{FMT}}$ via FMT.SimDecrypt. It sends this simulated secret key to $\mathcal{A}$ instead of an honestly generated one. This procedure is repeated via a hybrid argument for each PourTicket interaction that $\mathcal{A}$ engages in. If $\mathcal{A}$ distinguishes $\mathcal{G}_3$ from $\mathcal{G}_4$ with non-negligible advantage, then by the hybrid argument $\mathcal{A}$ must distinguish at least one set of intermediate hybrids with non-negligible advantage. However, this would imply that $\mathcal{A}'$ succeeds in distinguishing the distributions of Claim A.8 with non-negligible advantage, which contradicts the assumption. □

Lemma C.4. *No efficient adversary $\mathcal{A}$ can distinguish between the distributions $\mathcal{G}_5$ and $\mathcal{G}_6$.*

Proof. The proof relies on the strong unforgeability of the underlying one-time signature scheme SIG. The proof proceeds as follows. If $\mathcal{A}$ distinguishes between $\mathcal{G}_5$ and $\mathcal{G}_6$, then it must be the case that at some point $\mathcal{A}$ succeeds with non-negligible probability at producing a pour-ticket transaction that contains a tuple $S_{\mathsf{forge}} := (\mathsf{pk}_{\mathsf{SIG}}, m_{\mathsf{SIG}}, \sigma)$ with the conditions that (1) $\mathsf{pk}_{\mathsf{SIG}}$ was generated by an honest party, (2) $(m_{\mathsf{SIG}}, \sigma)$ was not produced by the honest party, and (3) $\mathsf{SIG.Verify}(\mathsf{pk}_{\mathsf{SIG}}, m_{\mathsf{SIG}}, \sigma) = 1$ as enforced by the VerifyTransaction algorithm. Given such a $\mathcal{A}$, we construct an adversary $\mathcal{A}'$ that breaks the strong unforgeability of SIG as follows.

First, $\mathcal{A}'$ runs the protocol as specified in $\mathcal{G}_5$, except that out of all of the PourTicket interactions initiated by honest parties, it selects one such interaction and obtains $\mathsf{pk}'_{\mathsf{SIG}}$ from the signing oracle instead of generating it as in the normal protocol interaction. It embeds this public key in spk sent to $\mathcal{A}$ in the first move of the protocol. If $\mathcal{A}$ does not abort the protocol, then $\mathcal{A}'$ uses the signing oracle to obtain σ' on the message m_{SIG}. (If $\mathcal{A}$ fails to complete the protocol, then $\mathcal{A}'$ instead queries the signing oracle on a random string of length $|m_{\mathsf{SIG}}|$.) Notice that $\mathcal{A}$'s view of this modified interaction is identical to that of $\mathcal{G}_5$. When $\mathcal{A}$ later appends a different pour-ticket transasction $\mathsf{tx}_{\mathsf{pt}}$ containg $(\mathsf{pk}_{\mathsf{SIG}}, m_{\mathsf{SIG}}, \sigma)$, $\mathcal{A}'$ halts if $\mathsf{pk}_{\mathsf{SIG}} \neq \mathsf{pk}'_{\mathsf{SIG}}$. Otherwise, $\mathcal{A}'$ outputs $(m_{\mathsf{SIG}}, \sigma)$ as a forgery in SIG's security experiment.

Note that if $\mathcal{A}$ succeeds in outputting $\mathsf{tx}_{\mathsf{pt}}$ with non-negligible probability, then $\mathcal{A}'$ will break the strong unforgeability of SIG with non-negligible advantage. □

Lemma C.5. *No efficient adversary $\mathcal{A}$ can distinguish between the distributions $\mathcal{G}_9$ and $\mathcal{G}_{10}$.*

Proof. The proof relies on the security of the *monotonic Merkle Trees* discussed in Section 7.2.6 and the underlying collision-resistant hash function $\mathsf{CRH}()$. The proof proceeds as follows. If $\mathcal{A}$ distinguishes $\mathcal{G}_9$ and $\mathcal{G}_{10}$, then it must be the case that $\mathcal{A}$ has, with non-negligible probability, succeeded in completing ℓ separate $\mathsf{PourTicket}$ interactions containing NIZK proofs $(\pi^1_{\mathsf{pt}}, \ldots, \pi^\ell_{\mathsf{pt}})$. Furthermore, once the knowledge extractor is applied to these proofs, each proof references the same deposit $\mathbf{d}$ and we obtain a set of witnesses $(\pi^1_{\mathcal{R}}, \ldots, \pi^\ell_{\mathcal{R}})$ and payment rates $(\mathfrak{w}^1_{\mathsf{r}}, \ldots, \mathfrak{w}^\ell_{\mathsf{r}})$ such that each witness is individually valid w.r.t. $\mathbf{d}$ and yet $\sum_{i=1}^{\ell} \mathfrak{w}^i_{\mathsf{r}} > \mathfrak{s}$, where $\mathfrak{s}$ is the sum that was extracted from $\mathbf{d}$ at the time it was minted. We show that such an $\mathcal{A}$ implies a second adversary $\mathcal{A}'$ that runs the experiment of $\mathcal{G}_9$ with $\mathcal{A}$ and succeeds in finding a collision for CRH with non-negligible probability.

Recall that each $\pi_{\mathcal{R}}$ is a path in a Merkle sum tree, which is constructed as follows: the values $\mathfrak{w}^1_{\mathsf{r}}, \ldots, \mathfrak{w}^\ell_{\mathsf{r}}$ are located at the leaves of the tree. Each internal node stores the sum $s = v_l + v_r$ of the values v_l, v_r of its children, and also stores a hash $h = \mathsf{CRH}(\mathsf{left} \| \mathsf{right} \| s)$ of the children and their sum. The NP statement for π_{pt} requires that the root of the tree and $\mathfrak{s}$ is embedded within the deposit $\mathbf{d}$, and therefore each path $\mathfrak{w}^i_{\mathsf{r}}$ terminates at the same root value and sum. We observe that if no collisions occur in the hash function CRH, the sum at the root of the tree must be $\geq \sum_{i=1}^{\ell} \mathfrak{w}^i_{\mathsf{r}}$.

Thus if $\mathcal{A}$ succeeds at distinguishing $\mathcal{G}_9$ and $\mathcal{G}_{10}$ with non-negligible advantage, then it must be the case that $\mathcal{A}'$ can extract two distinct proofs $\pi^i_{\mathcal{R}}$ and $\pi^j_{\mathcal{R}}$ that terminate in the same root value, yet at some point in the path reference distinct intermediate sums. However, this event can only occur if at some point in the two paths there exists a collision for the hash function CRH. The algorithm $\mathcal{A}'$ simply runs $\mathcal{A}$ until it obtains two such proofs. Upon extracting these proofs $\mathcal{A}'$ identifies the colliding messages (m_1, m_2) used as input to CRH and outputs this pair as a collision. In the event that $\mathcal{A}$ succeeds with non-negligible advantage at distinguishing the two games, then $\mathcal{A}'$ succeeds with non-negligible advantage in identifying a collision in CRH, which violates our assumption that CRH is collision resistant. □

References

[BBSU12] Simon Barber, Xavier Boyen, Elaine Shi, and Ersin Uzun. Bitter to better - how to make Bitcoin a better currency. In *Proceedings of the 16th International Conference on Financial Cryptography and Data Security*, FC '12, pages 399–414, 2012.

[BCG+14] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from Bitcoin. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, pages 459–474, 2014.

[Bit13] Bitcoinj. Working with micropayment channels. `https://bitcoinj.github.io/working-with-micropayments`, 2013.

[Blo14] Block Chain Analysis. Block chain analysis. `http://www.block-chain-analysis.com/`, 2014.

[BM89] Mihir Bellare and Silvio Micali. Non-interactive oblivious transfer and applications. In *Proceedings of the 9th Annual International Cryptology Conference*, CRYPTO '89, pages 547–557, 1989.

[BP15] Alex Biryukov and Ivan Pustogarov. Proof-of-work as anonymous micropayment: Rewarding a Tor relay. In *Proceedings of the 19th International Conference on Financial Cryptography*, FC '15, pages 445–455, 2015.

[BR99] Mihir Bellare and Ronald L. Rivest. Translucent cryptography - an alternative to key escrow, and its implementation via fractional oblivious transfer. *Journal of Cryptology*, 12(2):117–139, 1999.

[Bra93] Stefan Brands. Untraceable off-line cash in wallets with observers (extended abstract). In *Proceedings of the 13th Annual International Cryptology Conference*, CRYPTO '93, pages 302–318, 1993.

[Cal12] Mike Caldwell. Sustainable nanopayment idea: Probabilistic payments. `https://bitcointalk.org/index.php?topic=62558.0`, 2012.

[CEv87] David Chaum, Jan-Hendrik Evertse, and Jeroen van de Graaf. An improved protocol for demonstrating possession of discrete logarithms and some generalizations. In *Proceedings of the 6th Annual International Conference on Theory and Application of Cryptographic Techniques*, EUROCRYPT '87, pages 127–141, 1987.

[Cha82] David Chaum. Blind signatures for untraceable payments. In *Proceedings of the 2nd Annual International Cryptology Conference*, CRYPTO '82, pages 199–203, 1982.

[Cha15] Chainalysis. Chainalysis inc. `https://chainalysis.com/`, 2015.

[CHL05] Jan Camenisch, Susan Hohenberger, and Anna Lysyanskaya. Compact e-cash. In *Proceedings of the 24th Annual International Conference on Theory and Application of Cryptographic Techniques*, EUROCRYPT '05, pages 302–321, 2005.

[DDO+01] Alfredo De Santis, Giovanni Di Crescenzo, Rafail Ostrovsky, Giuseppe Persiano, and Amit Sahai. Robust non-interactive zero knowledge. In *Proceedings of the 21st Annual International Cryptology Conference*, CRYPTO '01, pages 566–598, 2001.

[DFKP13] George Danezis, Cédric Fournet, Markulf Kohlweiss, and Bryan Parno. Pinocchio Coin: building Zerocoin from a succinct pairing-based proof system. In *Proceedings of the 2013 Workshop on Language Support for Privacy Enhancing Technologies*, PETShop '13, 2013.

[DW15] Christian Decker and Roger Wattenhofer. A fast and scalable payment network with Bitcoin duplex micropayment channels. In *Proceedings of the 17th International Symposium on Stabilization, Safety, and Security of Distributed Systems*, SSS '15, pages 3–18, 2015.

[Elg85] Taher Elgamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31(4):469–472, 1985.

[Ell13] Elliptic. Elliptic enterprises limited. `https://www.elliptic.co/`, 2013.

[Fer93] Niels Ferguson. Single term off-line coins. In *Proceedings of the 12th Annual International Conference on Theory and Application of Cryptographic Techniques*, EUROCRYPT '93, pages 318–328, 1993.

[GGM16] Christina Garman, Matthew Green, and Ian Miers. Accountable privacy for decentralized anonymous payments. Cryptology ePrint Archive, Report 2016/61, 2016.

[GM16] Matthew Green and Ian Miers. Bolt: Anonymous payment channels for decentralized currencies. Cryptology ePrint Archive, Report 2016/701, 2016.

[Gol04] Oded Goldreich. *Foundations of Cryptography: Basic Applications*, volume 2. Cambridge University Press, New York, NY, USA, 2004.

[HAB+16] Ethan Heilman, Leen Alshenibr, Foteini Baldimtsi, Alessandra Scafuro, and Sharon Goldberg. TumbleBit: An untrusted Bitcoin-compatible anonymous payment hub. Cryptology ePrint Archive, Report 2016/575, 2016.

[HBHW16] Daira Hopwood, Sean Bowe, Tailor Hornby, and Nathan Wilcox. Zcash protocol specification, 2016. URL: https://github.com/zcash/zips/blob/master/protocol/protocol.pdf.

[HKZG15] Ethan Heilman, Alison Kendler, Aviv Zohar, and Sharon Goldberg. Eclipse attacks on Bitcoin's peer-to-peer network. In *Proceedings of the 24th USENIX Security Symposium*, Security '15, pages 129–144, 2015.

[HS12] Mike Hearn and Jeremy Spilman. Bitcoin contracts. https://en.bitcoin.it/wiki/Contract, 2012.

[KMS+16] Ahmed E. Kosba, Andrew Miller, Elaine Shi, Zikai Wen, and Charalampos Papamanthou. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *Proceedings of the 2016 IEEE Symposium on Security and Privacy*, SP '16, pages 839–858, 2016.

[LO98] Richard J. Lipton and Rafail Ostrovsky. Micropayments via efficient coin-flipping. In *Proceedings of the 2nd International Conference on Financial Cryptography*, FC '98, pages 1–15, 1998.

[MB15] Malte Möser and Rainer Böhme. Trends, tips, tolls: A longitudinal study of Bitcoin transaction fees. In *Proceedings of the 2nd Workshop on Bitcoin and Blockchain Research*, Bitcoin '15, pages 19–33, 2015.

[MGGR13] Ian Miers, Christina Garman, Matthew Green, and Aviel D. Rubin. Zerocoin: Anonymous distributed e-cash from Bitcoin. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, pages 397–411, 2013.

[Mic14] Silvio Micali. Universal payment systems. https://www.youtube.com/watch?v=xgA6TO7drok, 2014.

[MPJ+13] Sarah Meiklejohn, Marjori Pomarole, Grant Jordan, Kirill Levchenko, Damon McCoy, Geoffrey M. Voelker, and Stefan Savage. A fistful of Bitcoins: characterizing payments among men with no names. In *Proceedings of the 2013 Internet Measurement Conference*, IMC '13, pages 127–140, 2013.

[MR02] Silvio Micali and Ronald L. Rivest. Micropayments revisited. In *Proceedings of the Cryptographer's Track at the RSA Conference*, CT-RSA '02, pages 149–163, 2002.

[MRK03] Silvio Micali, Michael O. Rabin, and Joe Kilian. Zero-knowledge sets. In *Proceedings of the 44th Annual Symposium on Foundations of Computer Science*, FOCS '03, pages 80–91, 2003.

[Nak09] Satoshi Nakamoto. Bitcoin: a peer-to-peer electronic cash system, 2009. URL: http://www.bitcoin.org/bitcoin.pdf.

[NBF+16] Arvind Narayanan, Joseph Bonneau, Edward Felten, Andrew Miller, and Steven Goldfeder. *Bitcoin and Cryptocurrency Technologies: A Comprehensive Introduction*. Princeton University Press, 2016.

[Oka92] Tatsuaki Okamoto. Provably secure and practical identification schemes and corresponding signature schemes. In *Proceedings of the 12th Annual International Cryptology Conference*, CRYPTO '92, pages 31–53, 1992.

[PD16] Joseph Poon and Thaddeus Dryja. The Bitcoin lightning network: Scalable off-chain instant payments. https://lightning.network/lightning-network-paper.pdf, 2016.

[Ped91] Torben P. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In *Proceedings of the 11th Annual International Cryptology Conference*, CRYPTO '91, pages 129–140, 1991.

[PS15] Rafael Pass and Abhi Shelat. Micropayments for decentralized currencies. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security*, CCS '15, pages 207–218, 2015.

[PS16] Rafael Pass and Abhi Shelat. Micropayments for decentralized currencies. Cryptology ePrint Archive, Report 2016/332, 2016.

[RH11] Fergal Reid and Martin Harrigan. An analysis of anonymity in the Bitcoin system. In *Proceedings of the 3rd IEEE International Conference on Privacy, Security, Risk and Trust (PASSAT), and the 3rd IEEE International Conference on Social Computing (SocialCom)*, SocialCom/PASSAT '11, pages 1318–1326, 2011.

[Riv97] Ronald L. Rivest. Electronic lottery tickets as micropayments. In *Proceedings of the 1st International Conference on Financial Cryptography*, FC '97, pages 307–314, 1997.

[Riv04] Ronald L. Rivest. Peppercoin micropayments. In *Proceedings of the 8th International Conference on Financial Cryptography*, FC '04, pages 2–8, 2004.

[RKS15] Tim Ruffing, Aniket Kate, and Dominique Schröder. Liar, liar, coins on fire!: Penalizing equivocation by loss of Bitcoins. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security*, CCS '15, pages 219–230, 2015.

[RS13] Dorit Ron and Adi Shamir. Quantitative analysis of the full Bitcoin transaction graph. In *Proceedings of the 17th International Conference on Financial Cryptography and Data Security*, FC '13, pages 6–24, 2013.

[Sah99] Amit Sahai. Non-malleable non-interactive zero knowledge and adaptive chosen-ciphertext security. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, FOCS '99, pages 543–553, 1999.

[Sch91] Claus-Peter Schnorr. Efficient signature generation by smart cards. *Journal of Cryptology*, 4(3):161–174, 1991.

[Sha79] Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.

[SJ00] Claus-Peter Schnorr and Markus Jakobsson. Security of signed Elgamal encryption. In *Proceedings of the 6th Annual International Conference on the Theory and Application of Cryptology and Information Security*, ASIACRYPT '00, pages 73–89, 2000.

[ST99] Tomas Sander and Amnon Ta-Shma. Auditable, anonymous electronic cash extended abstract. In *Proceedings of the 19th Annual International Cryptology Conference*, CRYPTO '99, pages 555–572, 1999.

[vOR+03] Nicko van Someren, Andrew M. Odlyzko, Ronald L. Rivest, Tim Jones, and Duncan Goldie-Scot. Does anyone really need micropayments? In *Proceedings of the 7th International Conference on Financial Cryptography*, FC '03, pages 69–76, 2003.

[Whe96] David Wheeler. Transactions using bets. In *Proceedings of the 1996 International Workshop on Security Protocols*, SPW '96, pages 89–92, 1996.

[Wil14] Zak Wilcox. Proving your Bitcoin reserves. `https://iwilcox.me.uk/2014/proving-bitcoin-reserves`, 2014.

[Yao77] Andrew Chi-Chih Yao. Probabilistic computations: Toward a unified measure of complexity. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, FOCS '77, pages 222–227, 1977.