Flexible Anonymous Transactions (Flax): Towards Privacy-Preserving and Composable Decentralized Finance

Wei Dai*
Anoma and UC San Diego
weidai@eng.ucsd.edu

November 23, 2021

Abstract

Decentralized finance (DeFi) refers to interoperable smart contracts running on distributed ledgers offering financial services beyond payments. Recently, there has been an explosion of DeFi applications centered on Ethereum, with close to a hundred billion USD in total assets deposited as of September 2021. These applications provide financial services such as asset management, trading, and lending. The wide adoption of DeFi has raised important concerns, and among them is the key issue of *privacy*—DeFi applications store account balances in the clear, exposing financial positions to public scrutiny.

In this work, we propose a framework of anonymous and composable DeFi on public-state smart contract platforms. First, we define a cryptographic primitive called a flexible anonymous transaction (Flax) system with two distinctive features: (1) transactions authenticate additional information known as "associated data" and (2) transactions can be applied *flexibly* via a parameter that is determined at processing time, e.g. during the *execution time* of smart contracts. Second, we design an *anonymous* token standard (extending ERC20), which admits *composable* usage of anonymous funds by other contracts. Third, we demonstrate how the Flax token standard can realize privacy-preserving variants of the Ethereum DeFi ecosystem of today—we show contract designs for asset pools, decentralized exchanges, and lending, covering the largest DeFi projects to date including Curve, Uniswap, Dai stablecoin, Aave, Compound, and Yearn. Lastly, we provide formal security definitions for Flax and describe instantiations from existing designs of anonymous payments such as Zerocash, RingCT, Quisquis, and Zether.

*Work done in part while at NTT Research.

Contents

1 Introduction

Decentralized finance (henceforth DeFi) refers to composable smart contract applications providing financial services beyond payments, such as trading and lending. The DeFi sector has seen extraordinary growth in recent years, especially on Ethereum where protocols have close to $100 billion total value locked (TVL[1]) as of September 2021 [Pul]. Ethereum DeFi (cf. [Sch21] and [WPG+21] for good summaries) is an expansive ecosystem of heterogeneous yet interoperable smart contracts, created and operated from a diverse set of entities. Most of the current DeFi applications span across three distinct categories: asset management, decentralized exchanges (DEXs), and lending. Asset management applications, such as Yearn [Yea], work similar to managed funds or exchange-traded funds (ETFs) in traditional finance. DEXs, such as Uniswap [AZR] and Curve [Ego], allow trading of assets without interaction with counterparties. Lending protocols such as Aave [Aav], Compound [LH], and Dai stablecoin [Tea] allow users to deposit assets as collateral to borrow other assets, enabling "longing" and "shorting." Underlying these applications is a key building block for the ecosystem—the *token standard* ERC20 [VB], which allows contracts to transfer tokens using a standardized interface. There are now close to half a million distinct ERC20 tokens with more than fifty reaching a billion USD in market capitalization [Eth]. Unsurprisingly, many concerns have surfaced during the rise of DeFi (see [AEF+21] for a survey): scalability, contract vulnerabilities [LCO+16, QZLG20], front running [DGK+20], legal compliance, and privacy, just to list a few.

DeFi and privacy. Privacy is a fundamental challenge in DeFi. Indeed, the exact financial information of all transactions as well as the overall financial positions (assets and liabilities) of all users are public in the current ecosystem. Non-existent privacy is one of the main downsides of DeFi compared to traditional finance, where financial records are mostly kept confidential from public scrutiny. Let us examine the issue of privacy in more detail. There are two notions of privacy of interest for transactions: *anonymity* and *confidentiality*. Anonymity says that the originator of the transaction should be hidden. Confidentiality says that the financial information of each transaction should be hidden.

Unfortunately, *confidentiality is impossible* for smart-contract-based DeFi applications due to functionality requirements [AEC21]. DeFi contract states often involve financial information that is required for functionality, such as application reserve amounts or exchange rates. DeFi transactions change these publicly accessible values and hence reveal the transactions values. However, what we could hope to achieve is transaction *anonymity*. Indeed, the functionality of a DeFi transaction does not depend on the transaction owner. For example, a smart contract giving interfaces for trading token A with token B does not need information regarding the exact identity of the transaction originator, but only the values of tokens desired to be exchanged. We remark that transaction anonymity enhances the privacy of individuals, since the financial positions of each party cannot be traced exactly. Hence, we use the term anonymity when referring to transactions but the term *privacy-preserving* when referring to the overall system that admits anonymous transactions.

Privacy-preserving payments. There have been numerous constructions of anonymous and confidential cryptocurrencies. ZCash, originated from the work of Zerocash [BCG+14], and Monero, whose underlying system is called RingCT [Noe15, SALY17, YSL+20], are two widely deployed solutions. Other constructions exist such as Quisquis [FMMO19] and Zether [BAZB20, Dia20]. These systems focus on public key, direct, and private payments[2], building towards a cryptographic abstraction called Decentralized Anonymous Payment (DAP) systems, first coined by [BCG+14].

[1] TVL is a measure similar to assets under management (AUM) in traditional finance.

[2] Other protocols deviating from public key direct payment also exist. For example, Mimblewimble [Jed, Poe, FOS19] drops the use of public keys and requires communication between the sender and receiver.

There are some recent efforts to push DAPs beyond payments. Manta [CXZ21] and SwapCT [EMP+21] extend the syntax of DAPs to add token trading. These solutions are *ad-hoc*, not composable, and does not generalize to other DeFi applications.

Privacy-preserving execution of smart contracts. Another line of work aims to provide privacy-preserving execution of smart contracts, such as Hawk [KMS+16], Arbitrum [KGC+18], Ekiden [CZK+19], zkay [SBG+19], Zexe [BCG+20], and Kachina [KKK21]. Hawk and Ekiden rely on additional trust assumptions. Arbitrum keeps computation and data off-chain to achieve privacy. Zexe extends Zerocash with scripting capabilities and enables applications such as peer-to-peer asset exchanges. Kachina and zkay provides tools for constructing general-purpose privacy-preserving smart contracts. However, it remains to be shown whether these solutions can support *composability* comparable to Ethereum DeFi. Indeed, the "UTXO"-style computational model of Zexe deviate significantly from standard smart contracts. While zkay supports privacy-preserving tokens, the feasibility of other DeFi applications is not demonstrated. The latest promising development of Kachina does not yet fully support interaction between smart contracts.

Mixers and their short-comings. Sitting between privacy-preserving payments and privacy-preserving computation is a concept called a *mixer* (also known as a *tumbler*). They are, in the most general form, privacy-preserving payment systems where transactions can give "public" outputs, where funds are "un-shielded" and transferred to a regular blockchain account, which can then be used to interact with the existing blockchain ecosystem arbitrarily. Some notable mixers include CoinJoin [Max] (for Bitcoin), Möbius [MM18], and Tornado Cash [PSS] (for Ethereum). In fact, mixers such as Tornado Cash already enable the creation of ephemeral and anonymous accounts to interact with existing Ethereum DeFi protocols that grant users a good level of anonymity. However, doing so is opt-in, ad-hoc, and cumbersome for the user.

To elaborate, assuming a user has some assets in a mixer such as Tornado Cash, the user could (1) withdraw their asset into a new Ethereum address, (2) do the necessary DeFi transactions from the newly created address before (3) consolidating the output assets back to the mixer. Such workflows are indeed possible and provide anonymity aimed for in our work, at the expense of multiple complex steps that require explicit action from the user. We ask whether the above workflow can be re-designed so that the sequence of three operations are (a) *done without temporary addresses*, yet (b) *support arbitrarily complex, possibly multi-stage, smart-contract interactions between un-shield and re-shield*. Achieving this allows us to use such a "mixer" as a blockchain-native and anonymous token standard (counterpart to ERC20) that is built-in instead of opt-in.

Towards privacy-preserving and composable DeFi. A privacy-preserving DeFi ecosystem should also allow *composability*, enabling smart contracts to utilize the functionality of each other. Moreover, an ideal design should allow migration of existing DeFi applications. Similar to how ERC20 is central to DeFi, we believe that the key is an *anonymous token standard*. In particular, we ask:

*Can we design **anonymous** token standards, similar to ERC20?*

We answer this question affirmatively, showing that with slight modifications to existing syntax and constructions of DAPs, as well as careful design of an anonymous token standard with smart-contract platform support, we can replicate the Ethereum DeFi ecosystem in a privacy-preserving manner. We move on to highlight two key design challenges.

Design challenge one: anonymous authorization of token use. ERC20 utilizes the notion of "allowance" for users to authorize contracts to use their tokens. For example, a user may invoke "TokenContract.approve(SpenderContract, amount)" to approve the usage of upto "amount" tokens (issued by TokenContract) to be used by SpenderContract. TokenContract keeps track of

the allowance table between spenders and users to make sure that all spendings are previously authorized. Replicating this mechanism directly to anonymized accounts is problematic. Indeed, Zether designs a "locking" mechanism that is directly parallel to the notion of "allowance". An account can lock all of their tokens to a contract, restricting usage of their tokens to exactly one contract. However, unlike the transparent allowance system of ERC20, "locking" on Zether locks *the entire anonymity set* of the transaction. We want a authorization mechanism without the use of temporary addresses or the need for locking.

Our solution is a new mechanism that differs from the allowance-based spending of ERC20. In particular, we let transactions authorize "intents", which are smart-contract invocations with partially-fixed parameters. Looking ahead, spend transactions in our work are called "debit transactions", which authorize their exact call intent via a partial smart contract call. Token contracts must then verify the correct execution of the call intents before processing any spend transactions.

Design challenge two: "flexible" output amount. There are two features of output assets from DeFi interactions of concern: (1) the exact amount of output assets of a DeFi interaction depends on global contract states at execution time of the transaction, which are unknown to the user at transaction preparation time (e.g. a trade in Uniswap), and (2) a DeFi interaction can result in possible future output assets for the user that are processed at an arbitrary time in the future depending on contract logic (e.g. a debt repayment in a Dai stablecoin vault).

Our work proposes the notion of *flexible credit* transactions to facilitate the above two features. To support the first feature, it is easy to rely on the additive homomorphic property of the value commitment of DAPs (often ElGamal encryption) to allow the exact value change to be determined at execution time. To realize the second feature, we need to allow *credit transactions*, which are transactions that only increase the balances of users, to be valid in any future time once they are generated. This again comes essentially for free for existing DAPs.

1.1 Outline for the rest of the paper

Pin-pointing the right cryptographic abstraction. We propose FLexible Anonymous transaction (tX), or Flax for short (Section 2). Flax is an extension to the previous abstraction of DAPs with two distinct additions. First, we add associated data to transactions, which allows transactions to specify and authenticate their intended usage. Second, we add *flexible debit and credit* transactions, which can take an additional parameter at processing time, deferring the exact value change to be determined at the execution time of smart contracts.

Anonymous token standard. We use Flax to design a token standard (Section 3), which serves as the fabric of privacy-preserving DeFi[3]. Roughly, the Flax token standard is an extension to ERC20 with an additional mechanism to allow atomic authorization of token use and smart-contract executions. In particular, Flax transactions authenticate their intended usage, i.e. initiating contract calls and arguments, in their associated data, which is then verified by the token contract at processing time using read access to the call stack.

DeFi showcase. In Section 4, utilizing the Flax token standard, we discuss designs for asset pools, decentralized exchanges, and lending. In particular, we show how pillars of Ethereum DeFi such as Yearn [Yea], Uniswap [AZR], Curve [Ego], Dai stablecoin [Tea], Aave [Aav], and Compound [LH], can migrate from ERC20 to the Flax token standard with little changes.

Security and instantiations. We provide code-based security definitions for Flax in Section 5. In Section 6.1, we discuss instantiations of Flax from Zerocash, RingCT, and Quisquis. In Sec-

[3]Flax fiber is used in the making of linen, a fabric known for its "strength, coolness, and luster," according to the Merriam-Webster dictionary [MWD].

tion 6.2, we give a detailed construction $\text{FLAX}_{\text{Zether}}$ from Zether and provide a concrete-security treatment.

2 Flexible Anonymous Transaction Systems

In this section, we introduce our main cryptographic building block—FLexible Anonymous transaction (tX) system, or FLAX for short. FLAX is a cryptographic system providing interfaces for maintaining account balances as well as generation and processing of transactions to update account balances.

FLAX extends the syntax and functionality of DAPs, or Decentralized Anonymous Payments, first coined and studied by Zerocash [BCG+14], as well as follow-up works [FMMO19, BAZB20, Dia20, YSL+20, FOS19]. FLAX transactions enjoy a set of additional properties, which we detail in this section, that enables them to serve as the crucial fabric of *composable* financial smart contracts.

A transaction system FLAX consists of a parameter generation algorithm `ParamGen`, ledger algorithms `Setup`, `Process` as well as user algorithms `Kg`, `Read`, `CreateTx`. A parameter param is sampled from `ParamGen` and fixed globally.[4] We assume that all algorithms have access to param. Throughout, we fix an integer MAX representing the maximum value allowed in a transaction system (e.g. $\mathsf{MAX} = 2^{32} - 1$). Ledger algorithms behave as follows.

- $\mathtt{st} \leftarrow\!\!\$\ \mathtt{Setup}(\mathtt{lid})$ takes input a ledger identifier $\mathtt{lid} \in \{0,1\}^*$ and returns an initial state. We assume that the ledger identifier is stored as $\mathtt{st.lid}$.
- $\mathtt{st}'/\bot \leftarrow\!\!\$\ \mathtt{Process}(\mathtt{st}, \mathtt{tx})$. Algorithm `Process` takes input the ledger state `st`, a transaction `tx`, and applies the transaction `tx` to the current state `st` to return a new ledger state $\mathtt{st}'$. It returns $\bot$ unless `tx` is well-formed.

User algorithms behave as follows.

- $(\mathrm{pk}, \mathrm{sk}) \leftarrow\!\!\$\ \mathtt{KeyGen}()$, generates a key pair.
- $\mathtt{bal}/\bot \leftarrow \mathtt{Read}(\mathtt{st}, \mathrm{sk})$, returns the account balance $\mathtt{bal} \in [0, \mathsf{MAX}]$.
- $\mathtt{ctx}/\mathtt{dtx}/\mathtt{ttx}/\bot \leftarrow\!\!\$\ \mathtt{CreateTx}(\mathtt{st}, \mathrm{sk}, (\mathrm{pk}, \mathtt{amt}), \mathtt{val}, \mathtt{AD})$, generates a transaction. This algorithm creates three distinct types of transactions depending on the input structure, which consists of a ledger state `st`, a sender secret key sk, a receiver public key pk (possibly $\bot$), a to-be-hidden transfer amount $\mathtt{amt} \in [0, \mathsf{MAX}]$, a publicly-declared value $\mathtt{val} \in [-\mathsf{MAX}, \mathsf{MAX}]$, and some associated data $\mathtt{AD} \in \{0,1\}^*$.
  - ▷ When $\mathrm{pk} = \bot$ and $\mathtt{val} = 1$, the output is a credit transaction `ctx`, with $\mathtt{ctx.val} = 1$. Credit transactions can take an additional argument $k \in [0, \mathsf{MAX}]$ at processing time, denoting the exact amount that the ledger credits to the transaction owner (holder of sk).
  - ▷ When $\mathrm{pk} = \bot$ and $\mathtt{val} < 0$, the output in this case is a debit transaction `dtx`, with $\mathtt{dtx.val} = \mathtt{val}$. A debit transaction debits (subtracts) upto `val` tokens for the account associated with secret key sk. If the balance of sk is less than $-\mathtt{val}$, then an error value $\bot$ is returned. Debit transactions can take a parameter $k \in [\mathtt{val}, 0]$ (defaults to `val`) at processing time, denoted $\mathtt{dtx}[k]$, specifying the exact value to be debited.

[4] In applications, param is sampled and fixed by standards or protocol designers. In security proofs, it is sampled by the security game.

▷ Otherwise, when pk is specified, the output transaction is a transfer transaction, $\texttt{ttx}$. The transaction $\texttt{ttx}$ should change the originator account (of sk) by $-\texttt{amt} + \texttt{val}$, and the recipient account (of pk) by $\texttt{amt}$, resulting in a net change of $\texttt{val}$. If the balance of sk is less than $\texttt{amt} - \texttt{val}$, then an error value $\perp$ is returned. The transfer transaction does not take additional parameters at processing time.

We write $\texttt{CreateCreditTx}(\texttt{st}, \text{sk}, \texttt{AD})$, $\texttt{CreateDebitTx}(\texttt{st}, \text{sk}, \texttt{val}, \texttt{AD})$ to emphasize the exact type of transaction being generated. We use the notation $\texttt{tx}$ to refer to any transaction, and notations $\texttt{ttx}$, $\texttt{dtx}$, $\texttt{ctx}$ to refer to the particular types of transactions. We remark that we do not aim to hide transactions types in Flax. To reiterate, debit and credit transactions are *flexible*, or parametrizable, transactions. For example, $\texttt{dtx}[k]$ and $\texttt{ctx}[k]$ are transactions debiting and credit k tokens, respectively. For these transactions, the default parameter is assumed to be $k = \texttt{tx.val}$.

We usually assign many values to associated data, and for convenience we assume that $\texttt{AD}$ is a key-value map, and that their values are accessible as properties of the generated transaction. For example, suppose we set $\texttt{AD} := \{\mathsf{Key} : \mathsf{Value}\}$. Then, for any transaction $\texttt{tx}$ generated honestly using $\texttt{AD}$, it will hold that $\texttt{tx}.\mathsf{Key}$ is Value. We disallow $\texttt{val}$ being used as a key in $\texttt{AD}$, since $\texttt{val}$ is already a property of transactions.

Differences from DAPs. The syntax of Flax contains two distinctive differences from previous formalizations of DAPs. The first difference is that we add associated data that must be authenticated by the secret-key holder of the transactions. This allows us to have users specify the "intended" usage of transactions. Looking ahead, $\texttt{AD}$ is crucial in ensuring non-malleability of smart contract calls. The second difference is that Flax adds flexible debit and credit transactions, allow us to realize the functionality of "slippage" for DeFi transactions.[5]

2.1 Correctness and security

We describe the desired correctness and security properties of Flax informally here. Formal definitions, in the style of code-based games, as well as detailed discussions on differences from previous formulations, are given in Appendix 5.

Correctness. Correctness asks that users can generate valid credit transactions at any time as well as debit and transfer transactions if they have enough balance. Specifically, a user can run $\texttt{CreateCreditTx}$ regardless of their balance in the current ledger state, and the resulting transactions $\texttt{ctx}$ can be processed for *any* ledger state, i.e. $\texttt{Process}(\texttt{st}, \texttt{ctx}[k])$ returns a valid state for any $\texttt{st}$ and $k \in [0, \mathsf{MAX}]$. If a user has balance k in a ledger state $\texttt{st}$, i.e. $\texttt{Read}(\texttt{st}, \text{sk}) = k$, then he should be able to run $\texttt{CreateTx}$ that removes values up to k.

Consistency. Consistency provides security for the ledger, guaranteeing that $\texttt{tx.val}$ declares exactly the net value change incurred if $\texttt{tx}$ is to be processed. Specifically, the notion asks that *maliciously* generated transactions correctly declare their net value change. Looking ahead, consistency is usually ensured by the soundness of a proof system.

Transaction integrity. Transaction integrity guarantees security for honest users. It requires that an attacker, with active access to transaction interfaces of honest users, cannot generate *new* transactions that *decrease* the balance of the honest user. Intuitively, transaction integrity is very similar to notions such as strong unforgeability (SUF-CMA) of digital signatures and integrity of ciphertext (INT-CTXT) for authenticated encryption schemes.

[5] We note that slippage, or more generally transaction order malleability, is a double-edge sword. A transaction can be "front-run" [DGK+20] with high slippage.

FLAX from	Ledger		User		Authentication	
	Setup	$\lvert\mathtt{st}\rvert$	Read	Privacy	Mechanism	Structure size
Zerocash	Trusted	$\mathcal{O}(m)$	$\mathcal{O}(\lvert\mathtt{st}\rvert)$	Full	SNARK	$\mathcal{O}(c \cdot \log(m))$
RingCT	Pub. coin	$\mathcal{O}(m)$	$\mathcal{O}(\lvert\mathtt{st}\rvert)$	Tx ring	Ring Sig.	$\mathcal{O}(c \cdot r)$
Quisquis	Pub. coin	$\mathcal{O}(u)$	$\mathcal{O}(\lvert\mathtt{st}\rvert)$	Tx ring	NIZK	$\mathcal{O}(c \cdot r)$
Zether	Pub. coin	$\mathcal{O}(n)$	$\mathcal{O}(1)$	Acc ring	NIZK	$\mathcal{O}(r)$

Figure 1: Comparison of possible instantiations of FLAX from prior designs. We assume that there are n accounts, m overall transactions, and u unspent coins in the UTXO set (for practical systems $n \ll u \ll m$). For the authentication structure of transactions: r is the ring size and c is the coin overhead (number of coins required to realize the transaction amount). Running time of `Process` is proportional to the verification time of the authentication mechanism. Running time of `CreateTx` is proportional to the prover (or signing) time of the authentication mechanism. The concrete efficiencies of authentication mechanisms differ. For example, ring signatures can support larger ring sizes given the same computational budget compared to NIZKs.

Replay protection. The syntax of FLAX allows multiple concurrent ledger states in applications. It is paramount to prevent replay attacks both across and within instances. Replay protection guarantees that, for any transfer or debit transaction `tx`, it can only be processed *exactly once* by *exactly one* FLAX instance. Replay protection can be achieved generically without cryptographic assumptions. We can encode the intended ledger identifier as well as a transaction nonce in associated data of a transaction and verify them during processing time. Intuitively, the "heavy-lifting" is deferred to transaction integrity, which ensures that this information is authenticated.

Transaction privacy. There are two sub-notions of privacy for transactions: transaction anonymity and (transfer) transaction confidentiality. Transaction anonymity asks each transaction to hide the transaction originator (i.e. the secret key holder). Transaction confidentiality ensures that for transfer transactions, the amount `amt` is hidden from outside observers.

2.2 Instantiations

We describe instantiations of FLAX from existing DAPs schemes. Specifically, we look at Zerocash [BCG+14], RingCT [Noe15, SALY17, YSL+20], Quisquis [FMMO19], and Zether [BAZB20, Dia20]. Figure 1 gives a comparison of the instantiations. We describe the modifications necessary to realize the additional syntax required by FLAX, namely that (1) transactions authenticate associated data `AD` and (2) addition of flexible debit and credit transactions, with the latter required to be applicable to any state. Appendix 6.1 gives more details on UTXO-based instantiations and Appendix 6.2 gives a detailed instantiation $\text{FLAX}_{\text{Zether}}$ based on Zether.

Authenticating associated data. A common structure to all aforementioned systems is that transactions have a transaction body and a piece of authenticating information. For RingCT, the authentication mechanism is a ring signature. Hence, `AD` can simply be added to the signing message. For Zerocash, Quisquis, and Zether, the authentication mechanism is a non-interactive proof system. In particular, Zerocash uses a SNARK (e.g. [Gro16]), while Quisquis and Zether rely on NIZKs which are instantiated using the Fiat-Shamir transform, e.g. Bulletproofs [BBB+18]. For these systems, we can authenticate `AD` by embedding it into the instance, assuming that the proof system employed is (weakly) *simulation extractable* [Sah99, DDO+01, Gro06], which has known constructions [GM17, KLO20, BKSV21].

Credit and debit transactions for UTXO-based DAPs. The state of UTXO-based DAPs, such as Zerocash, RingCT, and Quisquis, contains a set of "coins." A transaction "spends" a set of input coins and creates a set of output coins. Each coin has an owner and encodes a value, usually as a homomorphic commitment to the value. The balance of each account is simply the sum of the values of coins owned. To realize flexible credit transactions, we can repurpose the "mint" functionality of these systems, i.e. a credit transaction simply creates a coin. Flexibility is possible due to the homomorphic property of the value commitment. Moreover, the validity of such coin does not depend on the ledger state and can be added to the UTXO-set at any time. Finally, debit transactions can be generically constructed by combining a transfer transaction with a zero transfer amount and a credit transaction.

Specialized construction from Zether. Zether differs from other DAPs in that it uses an account-based model. One could view Zether in the UTXO-model where each public key owns exactly one coin and repeated spending from the same coin is allowed. The balance of each public key, which is an ElGamal ciphertext, is stored at a distinct location in the state. Transactions are made anonymous via zero updates to decoys accounts (together with the originator and recipient, these accounts form the anonymity ring). It is not hard to modify their scheme to additionally give interfaces for `CreateDebitTx` and `CreateCreditTx`.

3 Flax Token Standard

We first give a brief overview of a smart contract platform (cf. [LCO+16] for a good overview). We also make clear a non-standard functionality required for the Flax token standard, which is read access to the inter-contract call stack and arguments. In Section 3.1, we present the Flax token standard and construction of smart contract calls involving Flax transactions. In Section 3.2, we describe a blockchain system that uses a native Flax token as the "gas token."

Smart contract platforms. A blockchain, for the purpose of our work, is a public state machine, whose transition functions and states are public. Smart contracts are deterministic programs that run on the blockchain. Smart contracts can invoke function calls to other smart contracts. We assume that each execution of a smart contract is initiated by an end-user via a *blockchain transaction* (not to be confused with a Flax transaction). Error encountered during execution of smart contracts *reverts* the execution, meaning any state changes for an invocation of a smart contract by an end-user only happen if the invocation finishes without any errors. In more detail, a blockchain transaction contains the intent of the transaction, specifying the contract and function to invoke as well as the call arguments. The transaction is then signed by the user preparing it, ensuring its non-malleability. Transactions in production systems often include funding information. On Ethereum, this is specified via the transaction gas price and gas limit. We remark that while user-funded transactions are common in current smart contract platforms, it is not necessarily the only model.

Required smart contract functionalities. Recall that smart contracts can invoke each other, and an execution starts from a user transaction in our model. We assume that the address of the caller is stored in a globally accessible variable `caller`. For example, if $\texttt{Contract}_A$ calls a $\texttt{Contract}_B.\texttt{Function}$, then inside the execution of $\texttt{Contract}_B$ function, we have $\texttt{caller} = \texttt{Contract}_A$. Additionally, we assume read access to the *full* inter-contract call stack and call arguments. For example, suppose a user calls $\texttt{Contract}_1.\texttt{Func}_1$, and each contract $\texttt{Contract}_i.\texttt{Func}_i$ invokes $\texttt{Contract}_{i+1}.\texttt{Func}_{i+1}$ for $i = [1..n-1]$. Then, during the execution of $\texttt{Contract}_n$ in this execution, each of the $n-1$ call frames, i.e. $\texttt{Contract}_i.\texttt{Func}_i(\text{arg}, \ldots)$ for $i \in [1..n-1]$ should be available for read access. We assume that the *internal* function calls of a contract are not included

Contract `TokenContract` extends ERC20

global `bal` // a table mapping of addresses to (public) balances

global `st` // a FLAX system state, keeping track (hidden) account balances

`Constructor`:

1 `st` ← FLAX.`Setup`(`this`)

`FTransfer`(`TX`): // `TX` is a set of FLAX transaction.

2 `netval` ← $\sum_{\texttt{tx}\in\texttt{TX}}$ `tx.val`

3 If (`netval` $\neq 0$) then // Net value change must be covered by the `caller`.

4 `bal`[`caller`] ← `bal`[`caller`] − `netval` ; Require (`bal`[`caller`] ≥ 0)

5 For `tx` ∈ `TX` do

6 `VerifyIntent`(`tx`) // Reverts if `tx.intent` does not match call trace

7 `st` ← FLAX.`Process`(`st`, `tx`) // Reverts if `tx` is in valid, i.e. `st` $= \perp$

Figure 2: Anonymous token standard. Each token contract maintains its own FLAX system. ERC20 interfaces enabling contracts to transfer tokens, such as `Transfer`, `Approve`, `TransferFrom` are not shown. Line 6 performs a matching between `tx.intent` and the current inter-contract call stack and call arguments. It reverts if the matching fails.

in the inter-contract call stack. For example, the inter-contract call stack does not change when $\texttt{Contract.Func}_1$ calls $\texttt{Contract.Func}_2$.

We remark that although the call stack requirement is not available for in-production platforms such as Ethereum[6], it is feasible to add such functionality. Indeed, an execution of smart-contracts inside a blockchain transaction is considered atomic and carried out as part of the blockchain state transition. Additionally, we think this feature can prove useful in other settings. For example, it can be used to prevent re-entry attacks [LCO+16].

3.1 Anonymous Token Standard and Transaction Intent

We first show how to build anonymous token contracts using FLAX. First, we briefly review the architecture of a token contract, and more in particular the ERC20 token standard [VB]. A token contract is a smart contract that (1) maintains account balances (2) exposes transaction interfaces to users and other smart contracts. Token contracts such as ERC20 store account balances in the clear for any address (either a user-owned address or a contract address). Our architecture differ in the following: we keep the standard functionality of a token contract (global state `bal`, as well as interfaces `Transfer`, `Approve` and `TransferFrom`), but add FLAX on top of it to enable privacy-preserving transactions.

The pseudocode of a token contract is given in Figure 2. The FLAX state `st` is initialized so that `st.lid` is equal to the contract address. Since contract states are public, we omit specifying a read interface for FLAX accounts and assume that users can obtain the contract state `st` to run FLAX.`Read` locally. The transfer interface `FTransfer` takes input a set of FLAX transactions. If the set of FLAX transactions has a non-zero net change in value, then it is ensured the net change is debited or credited to the `caller` (line 4). The intent of each FLAX transaction is first verified (line 6), before they are processed.

To demonstrate the purpose of the intent check at line 6, let us consider the following scenario. Suppose an honest user construct a transaction calling $\texttt{AMM.SwapAtoB}(\texttt{dtx}_A, \texttt{ctx}_B, \texttt{minOut})$ where

[6]The Ethereum virtual machine exposes the immediate caller, call data, the transaction originator, but not the entire call stack and call arguments.

$\texttt{dtx}_A$ is a debit transaction for $\texttt{Token}_A$. An attacker, after obtaining $\texttt{dtx}_A$, could misuse $\texttt{dtx}_A$ by constructing his own credit transaction $\texttt{ctx}'_B$ and calling $\texttt{AMM.SwapAtoB}(\texttt{dtx}_A, \texttt{ctx}'_B, \texttt{minOut})$, which transfers the output tokens to the attacker. More generally, a (not-yet processed) Flax transaction can be used to construct an adversarial blockchain transaction. To solve the problem, notice that the honest and adversarial calls to `AMM.SwapAtoB` differ in the credit transaction supplied, i.e. honest transaction calls $\texttt{AMM.SwapAtoB}(\texttt{dtx}_A, \texttt{ctx}_B, \texttt{minOut})$ while adversarial transaction swaps out $\texttt{ctx}_B$ with $\texttt{ctx}'_B$. Hence, to prevent such methods of misuse, we rely on (read-only) access to the inter-contract call stack.

Authenticating intent via the call stack. To solve the above problem, we let transactions specify their *intended* usage by specifying the call trace pattern in the field `tx.intent`. For example, $\texttt{dtx}_A\texttt{.intent}$ can be set to "$\texttt{AMM.SwapAtoB}(\star, \texttt{ctx}_B, \texttt{minOut})$", denoting that the intended initiating contract call is the function `SwapAtoB` of contract `AMM` and the last two arguments must be the credit transaction $\texttt{ctx}_B$ and the integer `minOut`. Each token contract *must* verify that the initiating call matches the one specified by the transaction to be processed. In pseudocode, we use `VerifyIntent` to denote such as check (line 6), i.e. `VerifyIntent` takes input `tx.intent` and compares the requirements specified against the read-only call stack contents. Although we only need `tx.intent` and `VerifyIntent` to specify and access the initiating contract call here, we state them in more generality. The exact implementation details of `tx.intent` and `VerifyIntent` are not important for our discussion here but a simple solution is to (1) fix a string encoding of the call stack and call arguments and (2) have `tx.intent` be a regular expression.

We now argue why the above check is sufficient in preventing the misuse of transactions. First, by replay protection (formal definition given in Appendix 5), each transfer and debit transaction can only be processed by one honest token. Hence, for an honestly constructed transaction `tx`, the only honest contract that deem `tx` valid is `tx.lid`. Next, at line 6 of contract `tx.lid`, the check $\texttt{VerifyIntent}(\texttt{tx.intent})$ is performed. For `tx` to be processed, it must have been that the initial call for this execution is consistent with `tx.intent`. Therefore, `tx` can only be processed via initiating contract call specified in `tx.intent`. We move on to discuss how client wallets can prepare Flax transactions in a general smart contract call.

Generating smart contract calls. To ensure that funds of users are used for their intended purpose, we encode all call arguments and credit transactions as part of `tx.intent` for transfer and debit transactions. In particular, consider a smart contract call of the form

$$\texttt{Contract.Function}(\texttt{tx}_1, \ldots, \texttt{tx}_n, \texttt{ctx}_1, \ldots, \texttt{ctx}_m, \mathrm{arg}_1, \ldots, \mathrm{arg}_k) \ ,$$

for some transfer or debit transactions $\texttt{tx}_1, \ldots, \texttt{tx}_n$, credit transaction $\texttt{ctx}_1, \ldots, \texttt{ctx}_m$, and other arguments. The user wallet will first prepare the credit transactions $\texttt{ctx}_1, \ldots, \texttt{ctx}_m$ with $\texttt{ctx}_i\texttt{.intent}$ fields set to "$\texttt{Contract.Function}(\star, \mathrm{arg}_1, \ldots, \mathrm{arg}_k)$", i.e encoding of the function call to `Function` of `Contract` with the last k call arguments specified. Next, the user generate $\texttt{tx}_1, \ldots, \texttt{tx}_n$ with $\texttt{tx}_i\texttt{.intent}$ field additionally specifying the newly generated credit transactions, i.e.

$$\text{“}\texttt{Contract.Function}(\star, \texttt{ctx}_1, \ldots, \texttt{ctx}_m, \mathrm{arg}_1, \ldots, \mathrm{arg}_k)\text{”} \ .$$

In Figure 3, we give an example smart contract invocation of an automated market-maker contract (described in Section 4) involving two types of tokens.

3.2 Blockchain design with native anonymous Flax tokens

While it is feasible to add anonymous transactions to any smart contract platform, doing so require the use of "relayers", i.e. third parties who submit contract calls on behalf of other users, if native blockchain transactions leak originating users. Indeed, this is proposed in Zether. While our

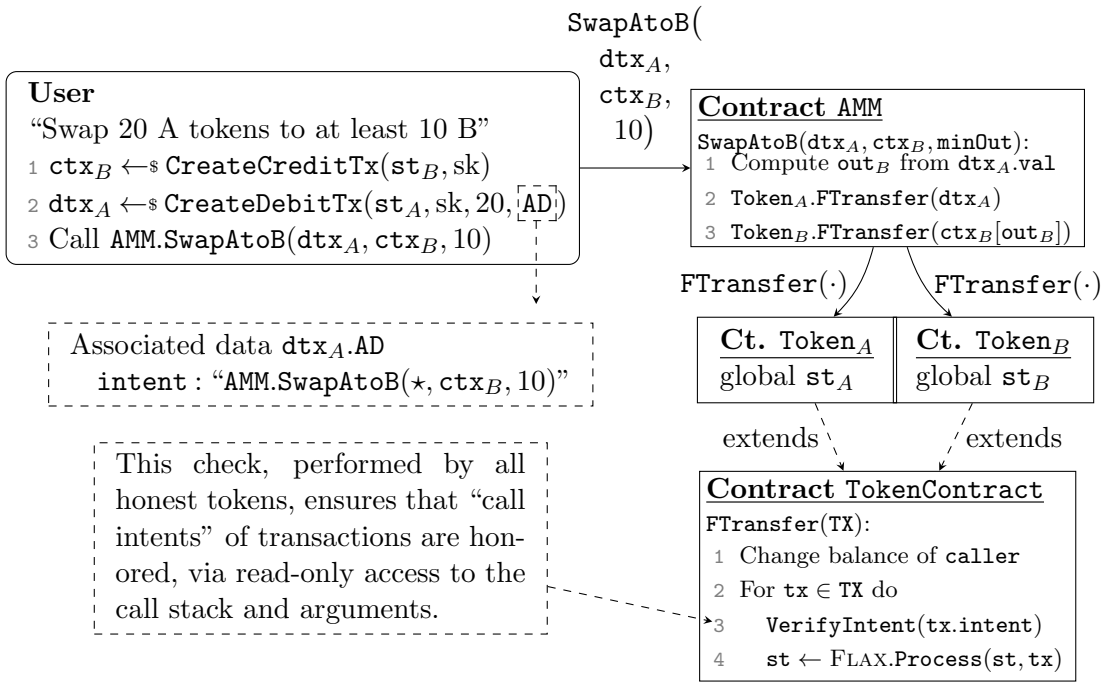

Figure 3: Example call to `AMM.SwapAtoB` initiated by a user. The user prepares their FLAX transactions and forms a smart contract call. All contracts are executed by a distributed ledger. User code is executed locally by the user. Dashed boxes contain explanations. The first lines of contracts `AMM` and `TokenContract` are abbreviated. The detailed contracts are given in Figures 2 and 5.

framework can be realized on any smart contract platform in the same manner, we will describe a system where a distinguished token contract is used as the native gas token.

Let us consider a distinguished contract $\texttt{Token}_{\text{FLAX}}$. We define a blockchain transaction to be a FLAX debit transaction $\texttt{dtx}_{\texttt{call}}$ with:

- $\texttt{dtx}_{\texttt{call}}.\texttt{val}$ specifying the maximum transaction fee.
- $\texttt{dtx}_{\texttt{call}}.\texttt{gasprice}$ specifying the gas price.
- $\texttt{dtx}_{\texttt{call}}.\texttt{intent}$ now specifies a full smart contract call, including all call arguments. For example, a value of "`Token.FTransfer(ttx)`".

In our special gas token contract, `VerifyIntent` can forego the checking of `intent`. Instead, the contract can process the intent of the transactions and debit the correct transaction fees as follows.

1 $\texttt{gasused} \leftarrow \texttt{Run}(\texttt{dtx}_{\texttt{call}}.\texttt{intent})$

2 $\texttt{st} \leftarrow \texttt{Process}(\texttt{st}, \texttt{dtx}_{\texttt{call}}[\texttt{dtx}_{\texttt{call}}.\texttt{gasprice} \cdot \texttt{gasused}])$

where `Run` is a special command executing the intent of the blockchain transaction (with a fresh inter-contract call stack) and returning the value of gas used (amount of computation performed).

4 Applications to Privacy-Preserving DeFi

In this section, we demonstrate how the FLAX-based token standard serves as the fabric of composable DeFi. In particular, we show contract designs for asset pools, automated market maker

Contract `Pool` extends `TokenContract`

`cptEnter`$(\texttt{val}_A, \texttt{val}_B)$ // Computes output pool tokens for input A and B tokens

`cptExit`$(\texttt{val}_{\texttt{Pool}})$ // Computes output tokens for $\texttt{val}_{\texttt{Pool}}$ pool tokens

`EnterPool`$(\texttt{dtx}_A, \texttt{dtx}_B, \texttt{ctx}_{\texttt{Pool}})$:

1 $(\texttt{in}_A, \texttt{in}_B, \texttt{out}_C) \leftarrow \texttt{cptEnter}(\texttt{dtx}_A.\texttt{val}, \texttt{dtx}_B.\texttt{val})$
2 $\texttt{Token}_A.\texttt{FTransfer}(\texttt{dtx}_A[\texttt{in}_A])$ // Transfer $\texttt{in}_A$ token A to this contract
3 $\texttt{Token}_B.\texttt{FTransfer}(\texttt{dtx}_B[\texttt{in}_B])$ // Transfer $\texttt{in}_B$ token B to this contract
4 $\texttt{bal}[\texttt{this}] \leftarrow \texttt{bal}[\texttt{this}] + \texttt{out}_{\texttt{Pool}}$ // Mint $\texttt{out}_{\texttt{Pool}}$ pool tokens
5 $\texttt{this}.\texttt{FTransfer}(\texttt{ctx}_{\texttt{Pool}}[\texttt{out}_{\texttt{Pool}}], \texttt{this})$ // Transfer out minted pool tokens

`ExitPool`$(\texttt{dtx}_{\texttt{Pool}}, \texttt{ctx}_A, \texttt{ctx}_B)$:

6 $(\texttt{out}_A, \texttt{out}_B) \leftarrow \texttt{cptExit}(\texttt{dtx}_{\texttt{Pool}}.\texttt{val})$
7 $\texttt{Token}_A.\texttt{FTransfer}(\texttt{ctx}_A[\texttt{val}_A])$ // Transfer $\texttt{out}_A$ token A out
8 $\texttt{Token}_B.\texttt{FTransfer}(\texttt{ctx}_B[\texttt{val}_B])$ // Transfer $\texttt{out}_B$ token B out
9 $\texttt{this}.\texttt{FTransfer}(\texttt{dtx}_{\texttt{Pool}})$ // Recover $\texttt{dtx}_{\texttt{Pool}}.\texttt{val}$ pool tokens
10 $\texttt{bal}[\texttt{this}] \leftarrow \texttt{bal}[\texttt{this}] - \texttt{dtx}_{\texttt{Pool}}.\texttt{val}$ // Burns $\texttt{dtx}_{\texttt{Pool}}.\texttt{val}$ pool tokens

Figure 4: Smart contract `Pool` implementing a contract-controlled and token-denominated fund. Note that `Pool` contract *extends* the token standard `TokenContract`, meaning it has global variables `bal`, `st`, as well as a public interface `FTransfer`.

(a.k.a liquidity pools), and collateralized debt positions. We remark that the design of these functionalities on ERC20 are well-known and in-production on Ethereum. For readers familiar with the ecosystem of Ethereum, we remark that we are checking the compatibility of the following applications against the Flax token standard: Yearn [Yea] (asset management), Uniswap [AZR] and Curve [Ego] (decentralized exchanges), Dai stablecoin [Tea], Aave [Aav] and Compound [LH] (lending). Our goal is to demonstrate that the Flax token standard is as flexible and usable as ERC20.

4.1 Anonymous token-denominated funds, a.k.a pools

As the first example, we describe a design of token-denominated funds. A fund, in context of finance, is a pool of money or assets allocated for a specific purpose. In our context, a fund, or more specifically a contract-controlled and token-denominated fund (which we also refer to as a pool) is a smart contract that (1) owns a set of assets (tokens), (2) issues outstanding shares (pool tokens), and (3) provides interfaces to exchange between underlying assets and pool shares. A pool can be seen as a DeFi counterpart to what is known in traditional finance as an exchange-traded fund (ETF).

Consider two underlying assets: token A and token B, hosted by contracts $\texttt{Token}_A$ and $\texttt{Token}_B$ respectively. We give the (abstract) contract implementing a pool holding token A and B in Figure 4. To enter, a user invokes `EnterPool` with two debit transactions $\texttt{dtx}_A, \texttt{dtx}_B$ with `val` specifying the max deposit values, as well as a credit transaction, $\texttt{ctx}_{\texttt{Pool}}$, for receiving the minted pool tokens. To exit, a user invokes `ExitPool` specifying a debit transaction, $\texttt{dtx}_{\texttt{Pool}}$, removing a desired amount of pool tokens, as well as two credit $\texttt{ctx}_A, \texttt{ctx}_B$ transactions for receiving the redeemed A and B tokens.

To fully specify a pool, one needs to specify functions that convert between values of asset and pool tokens, specifically `cptEnter` and `cptExit`. These functionalities can read the token reserves of the contract, but do not have any write access to global states—they simply compute and return token amounts. For example, a fund can be created between two tokens A and B such that each

Contract AMM extends Pool

cptAtoB($\texttt{val}_A$) // Computes amount of B tokens for input amount of $\texttt{val}_A$ A token

SwapAtoB($\texttt{dtx}_A, \texttt{ctx}_B, \texttt{minOut}$):

1 $\texttt{out}_B \leftarrow \texttt{cptAtoB}(\texttt{dtx}_A.\texttt{val})$; Assert ($\texttt{out}_B \geq \texttt{minOut}$)

2 $\texttt{Token}_A.\texttt{FTransfer}(\texttt{dtx}_A)$ // Transfer $-\texttt{dtx}_A.\texttt{val}$ to this contract

3 $\texttt{Token}_B.\texttt{FTransfer}(\texttt{ctx}_B[\texttt{out}_B])$ // Transfer $\texttt{out}_B$ token B out with $\texttt{ctx}_B$

Figure 5: Smart contract AMM implementing an anonymous automated market maker with liquidity pool token. Note that AMM *extends* the contract interface Pool and has additional interfaces EnterPool, ExitPool as well as FTransfer. Interface for swapping B to A is symmetrical and omitted.

pool token represents exactly 1 token A and 2 token B. In this scenario, $\texttt{cptEnter}(\texttt{val}_A, \texttt{val}_B)$ should return $(\texttt{val}, 2\texttt{val}, \texttt{val})$ to denote that the contract should take val token A, 2val token B, and output val pool tokens, where val is the largest value such that $\texttt{val} \leq \texttt{val}_A$ and $2\texttt{val} \leq \texttt{val}_B$. On the other hand, $\texttt{cptExit}(\texttt{val}_{\texttt{Pool}})$ simply returns $(\texttt{val}_{\texttt{Pool}}, 2\texttt{val}_{\texttt{Pool}})$, denoting that if $\texttt{val}_{\texttt{Pool}}$ tokens are burnt, then the contract can output $\texttt{val}_{\texttt{Pool}}$ token A and $2\texttt{val}_{\texttt{Pool}}$ token B. In general, these functions could return outputs that depend on the states of the contract and its token reserves. The exact implementation of these functions is left to the particular applications.

We remark that a Pool contract can use token A and B that it owns arbitrarily via the standard ERC20 interfaces. The interface shown here can be used as a basis to build asset management contracts, similar to asset vaults of Yearn.

4.2 Anonymous automated market-makers

An automated market maker (AMM) is a pool that additionally provides public interfaces to trade between its underlying assets. It is also known as a *liquidity pool* since it is a pool representing liquidity, i.e. trade-able assets.

Suppose there are two token contracts $\texttt{Token}_A$ and $\texttt{Token}_B$ hosting token A and B respectively. First, we recall a necessary functionality AMM.cptAtoB, which has read access to the amount of A and B tokens owned by the AMM contract and computes, for each input amount $\texttt{val}_A \in [0, \mathsf{MAX}]$ of token A, an output amount $\texttt{val}_B$ of token B that the AMM contract is willing to exchange for. We allow cptAtoB to return $\perp$, which indicates error and will revert the transaction invoking it. We note that there are many designs for the exact specification of cptAtoB, with the most popular being the constant product and constant sum formulas (used by Uniswap and Curve respectively). There are however other variants that give other trade-offs [AKC+19, AC20].

We present a construction of privacy-preserving AMM contract AMM in Figure 5. Due to the difference in transaction preparation and execution time, AMM contracts often let the user specify the *minimum* amount of output tokens, and the contract will revert the transaction if the exchange rate at execution time does not yield enough of the output token. The swap functionality supporting Flax transactions shall take a credit transaction $\texttt{ctx}_A$, a debit transaction $\texttt{dtx}_B$, and an integer minOut. Transaction $\texttt{ctx}_A$ is used to transfer an amount of token A to the public account of the AMM contract. Transaction $\texttt{dtx}_B$ is used to transfer the output amount of token B. Here, we crucially rely on the flexibility of a Flax transaction—the exact amount of token transferred is determined at the execution time of the smart contract call.

Contract CDP

global vault // List of all vaults

CheckLiqudationCondition(vid) → int/⊥ // Liquidation check

OpenVault($\texttt{dtx}_{A,\texttt{fund}}, \texttt{ctx}_{A,\texttt{refund}}, \texttt{ctx}_{B,\texttt{borrow}}, \texttt{borrow}_B$):

1 collateral ← $\texttt{dtx}_{A,\texttt{fund}}$.val // Amount of collateral A token

2 debt ← borrow // Amount of debt B token

3 $\texttt{Token}_A$.FTransfer($\texttt{dtx}_{A,\texttt{fund}}$) // Fund the vault with $\texttt{dtx}_{A,\texttt{fund}}$

4 $\texttt{Token}_B$.FTransfer($\texttt{ctx}_{B,\texttt{borrow}}$[borrow]) // Transfer out the borrow amount

5 Return vault.push((collateral, debt, $\texttt{ctx}_{A,\texttt{refund}}$)) // Returns vid

Repay($\texttt{dtx}_B$, vid):

6 (collateral, debt, $\texttt{ctx}_{A,\texttt{refund}}$) ← vault[vid]

7 $\texttt{Token}_B$.FTransfer($\texttt{dtx}_B$) // Transfer $\texttt{dtx}_B$.val to this contract

8 debt ← debt + $\texttt{dtx}_B$.val // Remove $\texttt{dtx}_B$.val from debt

9 If (debt ≤ 0) then // If debt is all paid, refund and delete vault

10 $\texttt{Token}_A$.FTransfer($\texttt{ctx}_{A,\texttt{refund}}$[collateral]) ; vault[vid] ← ⊥

11 Else vault[vid] ← (collateral, debt, $\texttt{ctx}_{A,\texttt{refund}}$) // Update debt value

Liquidate($\texttt{dtx}_B, \texttt{ctx}_A$, vid):

12 // Check if vault can be liquidated, and compute the output collateral

13 out ← CheckLiqudationCondition(vid) // Reverts if vault is healthy

14 (collateral, debt, $\texttt{ctx}_{A,\texttt{refund}}$) ← vault[vid]

15 Assert (debt = $-\texttt{dtx}_B$.val) // Liquidator must repay all debt

16 $\texttt{Token}_B$.FTransfer($\texttt{dtx}_B$) // Charge the liquidator in token B

17 $\texttt{Token}_A$.FTransfer($\texttt{ctx}_A$[out]) // Pay the liquidator in token A

18 // collateral − out is profit to the protocol

Figure 6: Smart contract implementing anonymous vault-based lending.

4.3 Anonymous vault-based lending

We describe a design for collateralized lending, or so-called collateralized debt positions, similar to Dai stablecoin, Aave, and Compound. For simplicity, we consider for now one type of collateral token (A), and one type of debt token (B).

Consider the CDP contract given in Figure 6. We utilize the notion of a vault, which represents collateral and debt. In particular, a vault consists of two integers and a Flax credit transaction, i.e. $(\texttt{collateral}, \texttt{debt}, \texttt{ctx}_{\texttt{refund}})$. Amount collateral of token A and debt of token B are stored in the clear. As a result, the overall amount of collateral token A and outstanding debt token B is known at all times to the smart contract. The beneficiary of the refund credit transaction $\texttt{ctx}_{\texttt{refund}}$ is the "owner" of the vault since this is the only entity that benefits from the repayment of debt.

Any user can anonymously create a vault via interface OpenVault, depositing collateral tokens and taking out any other supported tokens as debt. Anyone can repay a vault via Repay. A vault may be liquidated by other users if it becomes "under-collateralized", ensuring the solvency of the protocol. Typically, liquidation usually happens when the collateral to debt ratio (valued against some common token, say the dollar or the native token of the blockchain) falls below a certain threshold, e.g. 1.1. We encapsulate this liquidation check in a function CheckLiqudationCondition(vid), which either returns an error if the vault vault[vid] cannot be liquidated, or returns the amount of collateral for sale for a full liquidation of the vault.

Extensions to multi-asset and interest rates. We remark that the system designed here

closely resembles the CDP contract for the Dai stablecoin [Tea], where token B is owned by the CDP contract and minted for every borrow. However, lending can be done even if token B is external. To achieve this, we first need to allow token B as collateral to borrow token A, which completes the lending and borrowing market between token A and token B. Real-world protocols often set deposit and borrow interest rates for these markets as a function of the reserve ratio. As vault collateral and debt are stored in the clear, it is possible to accrue interest on them. For example, Compound [LH] keeps global variables, which accrues over time, recording the exact value of a single collateral or debt token. Furthermore, vault-based lending can be extended to multi-asset lending markets similar to Aave and Compound. However, we remark that CDP designed here crucially rely on the use of vaults, whereas Aave [Aav] and Compound [LH] allow borrowing without explicit creation of vaults.

5 Formal correctness and security notions

We formally define correctness and security notions in this section. We adopt code-based game-playing framework of [BR06]. A game $\mathbf{G}$ consists of a set of oracles and a main procedure, which usually invokes an adversary. To facilitate the legibility of definitions, we use $\mathsf{pkOf}(\mathrm{sk})$ to denote the public key associated with sk throughout this section.

Correctness. First, correctness asks that if $\mathtt{Read}(\mathtt{st}, \mathrm{sk}) = k$ for some ledger state $\mathtt{st}$ and secret key sk, then $\mathtt{CreateTx}(\mathtt{st}, \mathrm{sk}, \ldots)$ can be used to generate transactions transferring upto value k, i.e. transfer transactions with $\mathtt{amt} + \mathtt{val} \leq k$ and debit transactions with $\mathtt{val} \leq k$. Moreover, a user can at any time generate a credit transaction $\mathtt{ctx}$ that remain valid for any ledger state $\mathtt{st}$, meaning $\mathtt{Process}(\mathtt{st}, \mathtt{ctx}[k])$ returns a valid state for any k.

Correctness also asks that a correctly generated transaction transfer the correct amount. Specifically, let $\mathtt{ttx}$ be a transfer transaction generated via $\mathtt{CreateTx}(\mathtt{st}, \mathrm{sk}, (\mathrm{pk}', \mathtt{amt}), \mathtt{val})$. After processing, i.e. $\mathtt{st}' \leftarrow\!\!\$\ \mathtt{Process}(\mathtt{st}, \mathtt{ttx})$, we require the balance of the sender, i.e. $\mathtt{Read}(\cdot, \mathrm{sk})$, to decrease by exactly $\mathtt{amt} + \mathtt{tx.val}$. (We forego requiring the balance of the receiver to increase by $\mathtt{amt}$ here, as this property is implied by consistency and a definition would be cumbersome as we do not have the secret key of the receiver.) Similarly, consider a debit transaction $\mathtt{dtx}$ created with $\mathtt{CreateDebitTx}(\mathtt{st}, \mathrm{sk}, \mathtt{val})$. For any $k \in [0, \mathtt{val}]$, we require the balance of sk to decrease by k in the state change $\mathtt{st}' \leftarrow \mathtt{Process}(\mathtt{st}, \mathtt{dtx}[k])$. For credit transaction $\mathtt{ctx}$ created with $\mathtt{CreateCreditTx}(\mathtt{st}, \mathrm{sk})$, and any $k \in [0, \mathsf{MAX}]$, we require the balance of sk to increase by k in the state change $\mathtt{st}' \leftarrow \mathtt{Process}(\mathtt{st}, \mathtt{dtx}[k])$.

Consistency. Consistency guarantees security for the ledger. It asks that even maliciously generated transactions declare the correct amount of public net value change $\mathtt{tx.val}$. Formally, consider the following game $\mathbf{G}^{\mathrm{cons}}_{\mathrm{FLAX}}$.

Game $\mathbf{G}^{\mathrm{cons}}_{\mathrm{FLAX}}(\mathcal{A})$

$\mathsf{param} \leftarrow \mathtt{ParamGen}()$; $(\mathtt{lid}, (\mathtt{tx}_1, \ldots, \mathtt{tx}_n), \mathbf{Keys}) \leftarrow \mathcal{A}(\mathsf{param})$

$\mathtt{st} \leftarrow\!\!\$\ \mathtt{Setup}(\mathtt{lid})$; For $i \in [n]$ do $\mathtt{st} \leftarrow \mathtt{Process}(\mathtt{st}, \mathtt{tx}_i)$

Require $\big((\mathtt{st} \neq \bot)$ and $(\forall \mathrm{pk} \in \mathtt{st.pk} : \mathrm{pk} = \mathsf{pkOf}(\mathbf{Keys}[\mathrm{pk}]))\big)$

$\mathtt{val}_0 \leftarrow \sum_{i \in [n]} \mathtt{tx}_i.\mathtt{val}$; $\mathtt{val}_1 \leftarrow \sum_{\mathrm{pk} \in \mathtt{st.pk}} \mathtt{Read}(\mathtt{st}, \mathbf{Keys}[\mathrm{pk}])$

Return $(\mathtt{val}_0 \neq \mathtt{val}_1)$

Above, the adversary returns a ledger id, a sequence of transactions $\mathtt{tx}_1, \ldots, \mathtt{tx}_n$, as well as all secret keys for accounts involved, which is encoded in a table that maps public keys to secret keys. The adversary wins the game if the overall value of the final ledger state is *inconsistent* with

the sequence of transactions. We define the consistency advantage of adversary $\mathcal{A}$ against FLAX, $\mathbf{Adv}^{\text{cons}}_{\text{FLAX}}(\mathcal{A})$, to be the probability that $\mathcal{A}$ wins the game, i.e. $\Pr[\mathbf{G}^{\text{cons}}_{\text{FLAX}}(\mathcal{A})]$.

Transaction integrity. Transaction integrity guarantees security for honest users. It asks that the only transactions that can decrease the balance of an honest user are those directly generated by that user. Formally, consider game $\mathbf{G}^{\text{tx-int}}_{\text{FLAX}}$ given below.

Game $\mathbf{G}^{\text{tx-int}}_{\text{FLAX}}(\mathcal{A})$

1 $(\text{pk}, \text{sk}) \leftarrow\!\!\$\ \texttt{KeyGen}()$; $(\texttt{st}_0, \texttt{tx}^*) \leftarrow \mathcal{A}^{\text{CreateTx}}(\text{pk})$
2 Require $(\texttt{tx}^* \notin S)$; $\texttt{st}_1 \leftarrow \texttt{Process}(\texttt{st}_0, \texttt{tx}^*)$; Require $(\texttt{st}_1 \neq \bot)$
3 Return $(\exists \text{sk} \in \mathbf{Keys}.\texttt{values} : \texttt{Read}(\texttt{st}_1, \text{sk}) < \texttt{Read}(\texttt{st}_0, \text{sk}))$

NewUser:
4 $(\text{pk}, \text{sk}) \leftarrow\!\!\$\ \texttt{KeyGen}()$; $\mathbf{Keys}[\text{pk}] \leftarrow \text{sk}$; Return pk

CreateTx($\texttt{st}, \text{pk}, (\text{pk}', \texttt{amt}), \texttt{val}, \texttt{AD}$):
5 $\texttt{tx} \leftarrow\!\!\$\ \texttt{CreateTx}(\texttt{st}, \mathbf{Keys}[\text{pk}], (\text{pk}', \texttt{amt}), \texttt{val}, \texttt{AD})$; $S \xleftarrow{\cup} \texttt{tx}$; Return tx

Adversary $\mathcal{A}$ is given an honestly sampled public key pk and are given access to oracles that generate transactions using the associated secret key sk. Note that the adversary is allowed to supply any ledger state st in the input of these oracle calls. The returned transaction tx from each call is added to the set S. In the end, the adversary wins if it can produce a fresh transaction $\texttt{tx}^* \notin S$ that changes the balance for the honest user. We define the tx-int advantage of adversary $\mathcal{A}$ against FLAX, to be the probability that $\mathcal{A}$ wins the game, i.e. $\mathbf{Adv}^{\text{tx-int}}_{\text{FLAX}}(\mathcal{A}) := \Pr[\mathbf{G}^{\text{tx-int}}_{\text{FLAX}}(\mathcal{A})]$.

Replay protection. Replay protection guarantees that any (transfer or debit) transaction can only be processed once among any set of honest ledgers. Formally, consider the following game $\mathbf{G}^{\text{rep}}_{\text{FLAX}}$.

Game $\mathbf{G}^{\text{rep}}_{\text{FLAX}}(\mathcal{A})$

1 $\mathcal{A}^{\text{ProcessO}}()$; Return win

NewLedger(lid):
2 Require $(\texttt{lid} \notin L)$; Return $(L[\texttt{lid}] \xleftarrow{\cup} \texttt{Setup}(\texttt{lid}))$

ProcessO(lid, tx):
3 $\texttt{st} \leftarrow\!\!\$\ \texttt{Process}(L[\texttt{lid}], \texttt{tx})$; Require $(\texttt{st} \neq \bot)$
4 If $(\texttt{tx} \in S$ and $\texttt{tx.type} \neq \text{Credit})$ then $\texttt{win} \leftarrow \texttt{true}$
5 $S \xleftarrow{\cup} \texttt{tx}$; Return $(L[\texttt{lid}] \leftarrow \texttt{st})$

We define the rep-advantage of an adversary $\mathcal{A}$ to be the probability that $\mathcal{A}$ wins the security game, i.e. $\mathbf{Adv}^{\text{rep}}_{\text{FLAX}}\mathcal{A} = \Pr[\mathbf{G}^{\text{rep}}_{\text{FLAX}}(\mathcal{A})]$. We remark that replay protection can be added generically with no cryptographic assumptions. A simple construction is have each debit and transfer transfer specify a transaction nonce tx.nonce and the intended FLAX instance via tx.lid. In order to verify the transaction, a ledger state encodes its own ledger identifier as st.lid as well as set $\texttt{st}.S$ keeping track of all transaction nonces processed so far. We remark that constructions may have built-in methods to prevent replay protection, without the use of nonces. For example, the coin "serial number" in Zerocash effectively acts as a transaction nonce.

Transaction privacy. Privacy guarantees that honest transaction originators should be hidden (anonymity). Additionally, in the case that funds are being transferred between honest users, the transfer amount should also be hidden (confidentiality). To formalize the notion, we require

specification of a functionality, Anonymize, that computes the *compatible* anonymity information for a transaction input. It formally captures the uncertainty to an outside observer regarding the possible originators as well as receiver and hidden transaction amounts. Formally, Anonymize takes input a ledger state $\mathtt{st}$, a key value map $\mathbf{Keys}$ representing honest, and transaction input $(\mathrm{pk}_0, (\mathrm{pk}'_0, \mathtt{amt}), \mathtt{val}, \mathtt{AD})$. It outputs some $\mathrm{pk}_1, (\mathrm{pk}'_1, \mathtt{amt}')$ representing a compatible anonymizing input. We note that Anonymize could be trivial and only return the same sender, receiver, and transfer amount, in which case there is no privacy guarantee. Hence, the exact privacy guarantee depends on the Anonymize functionality specified. Consider the following game $\mathbf{G}^{\text{tx-priv}}_{\text{Flax,Anonymize}}$.

Game $\mathbf{G}^{\text{tx-priv}}_{\text{Flax,Anonymize}}(\mathcal{A})$

1 $b \leftarrow\$ \{0,1\}$; $\mathsf{param} \leftarrow\$ \mathtt{ParamGen}()$
2 $(\mathtt{st}, \mathrm{pk}_0, (\mathrm{pk}'_0, \mathtt{amt}_0), \mathtt{val}, \mathtt{AD}) \leftarrow\$ \mathcal{A}^{\text{NewUser},\text{CreateTx}}(\mathsf{param})$
3 Require $(\mathrm{pk}_0 \in \mathbf{Keys}.\mathtt{keys})$
4 $(\mathrm{pk}_1, \mathrm{pk}'_1, \mathtt{amt}'_1) \leftarrow\$ \mathsf{Anonymize}(\mathtt{st}, \mathbf{Keys}, \mathrm{pk}_0, (\mathrm{pk}'_0, \mathtt{amt}_0), \mathtt{val}, \mathtt{AD})$
5 $\mathtt{tx} \leftarrow\$ \mathtt{CreateTx}(\mathtt{st}, \mathbf{Keys}[\mathrm{pk}_b], (\mathrm{pk}'_b, \mathtt{amt}_b), \mathtt{val}, \mathtt{AD})$
6 $b' \leftarrow\$ \mathcal{A}(\mathtt{tx})$; Return $(b' = b)$

NewUser:
7 $(\mathrm{pk}, \mathrm{sk}) \leftarrow\$ \mathtt{KeyGen}()$; $\mathbf{Keys}[\mathrm{pk}] \leftarrow \mathrm{sk}$; Return pk

CreateTx$(\mathtt{st}, \mathrm{pk}, (\mathrm{pk}', \mathtt{amt}), \mathtt{val}, \mathtt{AD})$:
8 Require $(\mathrm{pk} \in \mathbf{Keys}.\mathtt{keys})$
9 Return $\mathtt{tx} \leftarrow\$ \mathtt{CreateTx}(\mathtt{st}, \mathbf{Keys}[\mathrm{pk}], (\mathrm{pk}', \mathtt{amt}), \mathtt{val}, \mathtt{AD})$

We define the priv-advantage of an adversary $\mathcal{A}$ against Flax and functionality Anonymize to be the probability that $\mathcal{A}$ predicts the correct bit, i.e. $\mathbf{Adv}^{\text{tx-priv}}_{\text{Flax,Anonymize}}(\mathcal{A}) := 2\Pr[\mathbf{G}^{\text{tx-priv}}_{\text{Flax}}(\mathcal{A})] - 1$.

5.1 Comparison with previous formulations

We note that since the syntax of DAPs studied by [BCG+14, FOS19, YSL+20, FMMO19, BAZB20] are all distinct from each other, their security definitions are also incompatible. Our formulation resemble that of Zerocash. However, we could not simply use their notions since their DAP syntax only allows transactions on coins rather than accounts (consisting of all owned coins).

Our notion of consistency is formalized as "balance" in Zerocash [SALY17, BCG+14] and RingCT [SALY17, YSL+20], "inflation resistance" in a study of Mimblewimble [FOS19]. Our notion of transaction integrity has been proposed in Zerocash [BCG+14] as transaction non-malleability, and as unforgeability in RingCT [YSL+20]. We remark Zether [BAZB20, Dia20] and Quisquis [FMMO19] do not formalize notions similar to transaction integrity. However, their notions of overdraft safety (or theft resistance in Quisquis) encompass some aspects of transaction integrity, but do not imply it. Intuitively, our notions of transaction integrity and replay protection (which can be obtained generically assuming the former) together guarantees strong forms of overdraft safety.

Notions of transaction privacy are studied under ledger indistinguishability in [BCG+14], transaction indistinguishability in [FOS19], anonymity in [FMMO19], and privacy in Zether [BAZB20, Dia20]. Our notion is both stronger and simpler since we let the adversary supply the ledger state.

6 Instantiations of Flax

6.1 Flax from UTXO-based DAPs

Review of UTXO DAPs. The most widely utilized paradigm in building DAPs is the UTXO approach. Some examples include Zerocash [BCG+14], RingCT [Noe15, SALY17, YSL+20], and Quisquis [FMMO19]. Their state contains a set of unspent-transaction outputs (UTXO), or "coins." These coins encode information regarding the coin owner (identified by public key), as well as their value. The value of the coin is usually encoded homomorphically (RingCT uses Pedersen commitment and Quisquis uses ElGamal encryption), or can be made so (in case of Zerocash). A transaction spends a set of coins (inputs) and generates a new set of coins (outputs). Double spending is prevented by ensuring that each coin can only be spent once. This is realized via coin serial numbers in Zerocash and key images in RingCT. In Quisquis, the entire set of input coins are deleted after the transaction is processed. Each transaction in these systems include (1) information regarding coins being spent (i.e. serial numbers, key images, or pointers to input coins), (2) set of newly created coins, and (3) a proof showing that the transaction preparer owns the coins being spent and that rules places on coins values are respected (e.g. net values are preserved). For (3) and in particular showing that the user holds the secret key for the input coins being spent, Zerocash employs a SNARK, RingCT relies on ring signatures, and Quisquis relies on Fiat-Shamir-based NIZKs. We refer to this as the *authentication mechanism*.

Authenticating associated data. To additionally authenticate associated data, we can rely on the authentication mechanism. This is simple for RingCT, since the mechanism is a ring signature. For Quisquis and Zerocash, we can rely on simulation extractable NIZKs and SNARKs. More specifically, since $\mathtt{AD}$ is encoded in the instances, we could rely on weak simulation extractability [Sah99, DDO+01, Gro06]. For NIZKs, although it is not known if NIZKs obtained via the Fiat-Shamir heuristic (e.g. Bulletproofs [BBB+18]) are simulation extractable, such result is known in a simpler setting of sigma protocols [FKMV12]. Moreover, generic transformations of obtaining simulation extractable NIZKs are known [DDO+01, Gro06, KZM+15]. For SNARKs, there are known efficient constructions [GM17, KLO20, BKSV21] that achieve weakly simulation extractability, with efficiency comparable to the benchmark construction of Groth16 [Gro16] when secret witnesses are short.

Credit and debit transactions. A credit transaction $\mathtt{ctx}$ is simply a transaction that creates a new coin. Flexibility is achieved by using the homomorphic property of the value commitment. Note that to process a credit transaction. The ledger only needs to verify that the coin indeed encodes a unit coin (of value 1), which does not depend on the state of the ledger. Hence, a credit transaction can be flexibly applied to any ledger state.

Debit from transfer and credit. We give of a generic way to construct flexible debit transactions using transfer and credit transactions. To generate a debit transaction, we first generate a credit transaction

$$\mathtt{ctx} \leftarrow\!\!\$\ \mathtt{CreateCreditTx}(\mathtt{st}, \mathrm{sk}, \mathtt{AD}) \ ,$$

then a transfer transaction

$$\mathtt{ttx} \leftarrow\!\!\$\ \mathtt{CreateTx}(\mathtt{st}, \mathrm{sk}, (\mathrm{pk}_0, 0), \mathtt{val}, \mathtt{AD}') \ ,$$

where pk_0 is a dummy public key (could be the public key for sk) and $\mathtt{AD}'$ is $\mathtt{AD}$ with an additional field $\mathtt{AD}'.\mathtt{ctx}$ set to the credit transaction $\mathtt{ctx}$ generated previously. This transfer transaction now represents a debit transaction $\mathtt{dtx}$. The transaction $\mathtt{dtx}[k]$ for some $k \in [-\mathtt{val}, 0]$ is processed as two transactions $\mathtt{ttx}$ and $\mathtt{ctx}[\mathtt{val} - k]$ (which must be processed atomically).

Other UTXO-based cryptocurrencies. Mimblewimble [Jed, Poe] is a UTXO-based "aggregate cash" [FOS19] that has a different architecture compared to that of Zerocash, RingCT, or Quisquis in that there are no public keys and payments require communication. While it is plausible that a Flax-like system could be built on top of Mimblewimble, we do not pursue it here.

Coin overhead. We remark that DAPs such as Zerocash, RingCT, and Quisquis only expose an interface to transact on a fixed set of coins. Since each public key can hold an arbitrary number of coins, transactions spending a large fraction of the balance of a public key could incur numerous sub-transactions. We do not explicitly model how a FLAX `CreateTx` call is mapped to coin spending transactions here.

6.2 Flax from Zether

In this section, we instantiate Flax using the construction of Zether [BAZB20] and Anonymous Zether [Dia20]. We first give the necessary background on groups and proof systems, as well as a review of Zether, before presenting our instantiation and analysis.

6.2.1 Preliminary

For the rest of the section, we fix a cyclic group $\mathbb{G}$ of prime order p. Consider the following two games capturing DL and q-DDH problems.

Game $\mathbf{G}^{\mathrm{dl}}_{\mathbb{G},g}(\mathcal{A})$

1 $x \leftarrow\!\!\$\ [p]$
2 $x' \leftarrow\!\!\$\ \mathcal{A}(g^x)$
3 Return $(x = x')$

Game $\mathbf{G}^{\mathrm{ddh}}_{\mathbb{G},g,q}(\mathcal{A})$

1 $b \leftarrow\!\!\$\ \{0,1\}$; $x \leftarrow\!\!\$\ [p]$; For $i \in [q]$ do $y_i \leftarrow\!\!\$\ [p]$
2 $v_0 \leftarrow\!\!\$\ \mathbb{G}^{2q+1}$; $v_1 \leftarrow (g^x, g^{y_1}, g^{xy_1}, \ldots, g^{y_q}, g^{xy_q})$
3 $b' \leftarrow\!\!\$\ \mathcal{A}(v_b)$; Return $(b = b')$

We define the DL and q-DDH advantage of an adversary $\mathcal{A}$ against to be $\mathbf{Adv}^{\mathrm{dl}}_{\mathbb{G},g}(\mathcal{A}) := \Pr[\mathbf{G}^{\mathrm{dl}}_{\mathbb{G},g}(\mathcal{A})]$, and $\mathbf{Adv}^{\mathrm{ddh}}_{\mathbb{G},g,q}(\mathcal{A}) := \Pr[\mathbf{G}^{\mathrm{ddh}}_{\mathbb{G},g,q}(\mathcal{A})]$, respectively.

Relations and proof systems. Let R be a relation. For pairs $(x, w) \in \mathsf{R}$, we call x the instance and w the witness. A non-interactive proof system Π for relation R consists of three algorithms `Setup`, P, and V. We require a zero-knowledge simulators simulator $\Pi.\mathsf{S}$, its associated setup $\Pi.\mathsf{SSetup}$, as well as an extractor $\Pi.\mathsf{Ext}$. We adopt the concrete-security treatment of [Bel20]. Consider the zero-knowledge game $\mathbf{G}^{\mathrm{zk}}_{\Pi}$ given below.

Game $\mathbf{G}^{\mathrm{zk}}_{\Pi}(\mathcal{A})$

1 $b \leftarrow\!\!\$\ \{0,1\}$
2 $(\mathsf{crs}_0, \mathsf{td}) \leftarrow\!\!\$\ \Pi.\mathsf{SSetup}$
3 $\mathsf{crs}_1 \leftarrow\!\!\$\ \Pi.\mathtt{Setup}$
4 $b \leftarrow\!\!\$\ \mathcal{A}^{\mathrm{P_F},\mathrm{Ex}}(\mathsf{crs}_b)$
5 Return $(b = b')$

$\mathrm{P_F}(x, w)$:

6 Require $(\mathsf{R}(\mathsf{crs}, x, w))$
7 $\pi_0 \leftarrow \Pi.\mathsf{S}(\mathsf{crs}_0, \mathsf{td}, x)$
8 $\pi_1 \leftarrow\!\!\$\ \Pi.\mathrm{P}(\mathsf{crs}_1, x, w)$
9 $Q \xleftarrow{\cup} (x, \pi_b)$
10 Return π_b

We define the zero-knowledge advantage of an adversary $\mathcal{A}$ to be $\mathbf{Adv}^{\mathrm{zk}}_{\Pi}(\mathcal{A}) = \Pr[\mathbf{G}^{\mathrm{zk}}_{\Pi}(\mathcal{A})]$. Next, we define the standard notion of soundness, which asks that only valid instances can have valid proofs. Formally, consider the game $\mathbf{G}^{\mathrm{snd}}_{\Pi}$ given below.

Game $\mathbf{G}^{\mathrm{snd}}_{\Pi}(\mathcal{A})$

1 $\mathsf{crs} \leftarrow\!\!\$\ \Pi.\mathtt{Setup}()$
2 $\mathcal{A}^{\mathrm{V_F}}(\mathsf{crs})$; Return win

$\mathrm{V_F}(x, \pi)$:

3 Require $(x \notin L_{\mathsf{R}})$
4 Return $(\mathsf{win} \leftarrow \Pi.\mathrm{V}(\mathsf{crs}, x, \pi))$

We define the soundness advantage of an adversary $\mathcal{A}$ against Π to be $\mathbf{Adv}^{\mathrm{snd}}_{\Pi}\mathcal{A} := \Pr[\mathbf{G}^{\mathrm{snd}}_{\Pi}(\mathcal{A})]$. We move on to define simulation extractablility [Sah99, DDO+01, Gro06]. Which is a strong notion requiring that the extractor to extract valid witnesses from forged proofs for an adversary even if the adversary has seen simulated proofs on possibly incorrect instances. Formally, consider the game $\mathbf{G}^{\mathrm{xt}}_{\Pi}(\mathcal{A})$ given below.

Game $\mathbf{G}^{\mathrm{xt}}_{\Pi}(\mathcal{A})$

1 $(\mathsf{crs}, \mathsf{td}) \leftarrow\!\!\$\ \Pi.\mathsf{SSetup}$
2 $\mathcal{A}^{\mathrm{Pr},\mathrm{Ex}}(\mathsf{crs})$
3 Return win

$\mathrm{Pr}(x)$:
4 $\pi \leftarrow \Pi.\mathsf{S}(\mathsf{crs}, \mathsf{td}, x)$
5 $Q \xleftarrow{\cup} (x, \pi)$
6 Return π

$\mathrm{Ex}(x, \pi)$:
7 Require $((x, \pi) \notin Q)$
8 Require $(\Pi.\mathrm{V}(\mathsf{crs}, x, \pi))$
9 $w \leftarrow\!\!\$\ \Pi.\mathsf{Ext}(\mathsf{crs}, \mathsf{td}, x, \pi)$
10 $\mathsf{win} \leftarrow \neg\mathsf{R}(\mathsf{crs}, x, w)$
11 Return win

We define the simulation extractable (XT) advantage of $\mathcal{A}$ against Π to be $\mathbf{Adv}^{\mathrm{xt}}_{\Pi}(\mathcal{A}) := \Pr[\mathbf{G}^{\mathrm{xt}}_{\Pi}(\mathcal{A})]$.

6.2.2 Review and Instantiating FLAX

We first recall the construction of Zether. Zether considers accounts $\mathsf{Acc} : \mathbb{G} \to \mathbb{G}^2$. Public key $\mathrm{pk} \in \mathbb{G}$ is mapped to an ElGamal ciphertext encrypting the balance $b \in [0, \mathsf{MAX}]$ as $\mathsf{Acc}(\mathrm{pk}) = (C, D) = (\mathrm{pk}^r g^b, g^r)$ for some $r \in \mathbb{Z}_p$ that is unknown.

Warm-up: confidential transactions. Suppose Alice, holding b coins under public key pk_A and secret key sk_A with current account state $\mathsf{Acc}(\mathrm{pk}_A) = (C_A, D_A)$, wants to send c coins to Bob, who has public key pk_B. The transaction $\mathtt{tx}$ consists of $(C_{\mathtt{tx},A}, C_{\mathtt{tx},B}, D_{\mathtt{tx}}, \pi)$ such that π is a proof certifying that Alice knows a witness $w = (\mathrm{sk}_A, r, c, b-c)$ to the instance $x = (g, \mathrm{pk}_A, \mathrm{pk}_B, C_A, D_A, C_{\mathtt{tx},A}, C_{\mathtt{tx},B}, D_{\mathtt{tx}})$ for the following relation:

$$\begin{aligned}\mathsf{R}_{\mathrm{ConfTransfer}} = \{\ (g, \mathrm{pk}_A, \mathrm{pk}_B, C_A, D_A, C_{\mathtt{tx},A}, C_{\mathtt{tx},B}, D_{\mathtt{tx}}\ ;\ \mathrm{sk}_A, r, c, b')\ :\ \\ g^{\mathrm{sk}_A} = \mathrm{pk}_A \wedge D = g^r \\ \wedge\ C_{\mathtt{tx},A} = \mathrm{pk}_A^r g^{-c} \wedge C_{\mathtt{tx},B} = \mathrm{pk}_B^r g^c \\ \wedge\ (C_A / C_{\mathtt{tx},A}) = (D_A / D)^{\mathrm{sk}_A} g^{b'} \\ \wedge\ c \in [0, \mathsf{MAX}] \wedge b' \in [0, \mathsf{MAX}]\ \}\ .\end{aligned} \tag{1}$$

To process this transaction, the smart contract verifies the proof π before simply setting the new state $(C'_A, D'_A), (C'_B, D'_B)$ to be

$$C'_A \leftarrow C_A \cdot C^{-1}_{\mathtt{tx},A}\ ;\ D'_A \leftarrow D_A \cdot D^{-1}_{\mathtt{tx}}$$
$$C'_B \leftarrow C_B \cdot C_{\mathtt{tx},B}\ ;\ D'_B \leftarrow D_B \cdot D_{\mathtt{tx}}\ .$$

Private transactions from anonymity set. Privacy can be added by hiding the actual sender and receiver behind a set of randomly selected users, which is called the *anonymity set*. To do this, Alice "hides" $\mathrm{pk}_A, \mathrm{pk}_B$ within a set $\mathbf{pk} = \{\mathrm{pk}_1, \ldots, \mathrm{pk}_n\}$ of public keys. The transaction now specifies n ElGamal ciphertexts, stored in a key-value map $\mathtt{tx}.\mathbf{C}$, under the same randomness $\mathtt{tx}.D$. A well-formed transaction needs to prove that the encrypted values are all zero besides two entries, and those two entries must satisfy the same constraints outlined for confidential transactions. However, in doing so, the system now has created a "front-running" problem, whereby a well-formed transaction $\mathtt{tx}$, with anonymity set $\mathbf{pk}$, is rejected if another transaction $\mathtt{tx}'$, whose anonymity intersects with $\mathbf{pk}$, gets processed before $\mathtt{tx}$. To get around this issue, Zether uses a *transaction nonce*, u which must be proved to be of the form $g^{\mathrm{sk}}_{\mathsf{epoch}}$ that is included as part of each transaction. Zether enforces that, during each epoch, nonces must not repeat.

An anonymous and confidential transfer of $\texttt{amt}$ coins from pk to pk$'$, with publicly declared output of $\texttt{val}$ coins is a transaction $\texttt{tx}$, consisting of fields $\texttt{tx}.u$, $\texttt{tx}.D$, $\texttt{tx.val}$, $\texttt{tx}.\mathbf{C}$, and $\texttt{tx.AD}$, whose validity is checked via the following relation.

$$\begin{aligned}\mathsf{R}_{\text{Transfer}} = \{\big((g, g_{\mathsf{epoch}}, \mathsf{Acc}', \texttt{tx});\ (\text{sk}, r, \texttt{amt}, b', \text{pk}, \text{pk}')\big) :\\ \texttt{tx.type} = \text{Transfer}\\ \wedge\, g^{\text{sk}} = \text{pk} \wedge g_{\mathsf{epoch}}^{\text{sk}} = \texttt{tx}.u \wedge \texttt{tx}.D = g^r\\ \wedge\, \forall \text{pk}'' \in (\texttt{tx}.\mathbf{pk} - \{\text{pk}, \text{pk}'\}) : \texttt{tx}.\mathbf{C}[\text{pk}''] = \text{pk}''^r\\ \wedge\, \texttt{tx}.\mathbf{C}[\text{pk}] = g^{-\texttt{amt}+\texttt{tx.val}}\text{pk}^r \wedge \texttt{tx}.\mathbf{C}[\text{pk}'] = g^{\texttt{amt}}\text{pk}'^r\\ \wedge\, \texttt{tx.val} \in [0, \mathsf{MAX}] \wedge \texttt{amt} \in [0, \mathsf{MAX}]\\ \wedge\, \frac{\mathsf{Acc}'[\text{pk}].C}{\texttt{tx}.\mathbf{C}[\text{pk}]} = g^{b'} \left(\frac{\mathsf{Acc}'[\text{pk}].D}{\texttt{tx}.D}\right)^{\text{sk}} \wedge b' \in [0, \mathsf{MAX}]\} \ .\end{aligned} \tag{2}$$

Note that $\texttt{tx.AD}$ is not used in the validity check of the relation above. However, it is crucially included as part of the instance to ensure that a proof authenticates $\texttt{tx.AD}$.

This concludes the review of Zether. We are now ready to state our additions.

Debit and credit transactions. We simplify the private transfer transactions to only update the balance of one account out of n. A debit transaction $\texttt{tx}$ consists of fields $\texttt{tx}.u$, $\texttt{tx.val}$, $\texttt{tx}.\mathbf{C}$, and $\texttt{tx}.D$, $\texttt{tx.type} = \text{Debit}$. The transaction $\texttt{tx}$ is valid if it is an instance of the following relation.

$$\begin{aligned}\mathsf{R}_{\text{Debit}} = \{\big((g, g_{\mathsf{epoch}}, \mathsf{Acc}', \texttt{tx});\ (\text{sk}, r, b', \text{pk})\big) :\\ \texttt{tx.type} = \text{Debit}\\ \wedge\, g^{\text{sk}} = \text{pk} \wedge g_{\mathsf{epoch}}^{\text{sk}} = \texttt{tx}.u \wedge \texttt{tx}.D = g^r\\ \wedge\, \forall \text{pk}' \in (\texttt{tx}.\mathbf{pk} - \{\text{pk}\}) : \texttt{tx}.\mathbf{C}[\text{pk}'] = \text{pk}'^r\\ \wedge\, \texttt{tx}.\mathbf{C}[\text{pk}] = g^{\texttt{tx.val}}\text{pk}^r \wedge \texttt{tx.val} \in [0, \mathsf{MAX}]\\ \wedge\, \frac{\mathsf{Acc}'[\text{pk}].C}{\texttt{tx}.\mathbf{C}[\text{pk}]} = g^{b'} \left(\frac{\mathsf{Acc}'[\text{pk}].D}{\texttt{tx}.D}\right)^{\text{sk}} \wedge b' \in [0, \mathsf{MAX}]\} \ .\end{aligned} \tag{3}$$

The above relation ensures that only one account out of set S is being subtracted by $-\texttt{val}$, and that the remaining balance of this account is positive after processing of this transaction.

Credit transactions have the benefit of not having to prove that the remaining balance is positive. Moreover, we can drop the transaction nonce. We consider transactions that increment exactly the balance of one account out of $\mathbf{pk}$. The validity of a credit transaction $\texttt{tx}$ is checked via the following relation.

$$\begin{aligned}\mathsf{R}_{\text{Credit}} = \{\big((g, \mathsf{Acc}', \texttt{tx});\ (\text{sk}, r, \text{pk})\big) :\\ \texttt{tx.type} = \text{Credit}\\ \wedge\, g^{\text{sk}} = \text{pk} \wedge \texttt{tx}.D = g^r \wedge \texttt{tx}.\mathbf{C}[\text{pk}] = g \cdot \text{pk}^r\\ \wedge\, \forall \text{pk}' \in (\texttt{tx}.\mathbf{pk} - \{\text{pk}\}) : \texttt{tx}.\mathbf{C}[\text{pk}'] = \text{pk}'^r \ .\end{aligned} \tag{4}$$

Instantiating the relation and proof system. First, we consider putting the relations for anonymous transfer, debit, and credit transactions together. Specifically, let

$$\mathsf{R} := \mathsf{R}_{\text{Transfer}} \cup \mathsf{R}_{\text{Debit}} \cup \mathsf{R}_{\text{Credit}} \ , \tag{5}$$

Relation R is the relation of interest for our FLAX instantiation. Note that instances of R are of

Scheme FLAX

ParamGen():

1 $\mathsf{crs} \leftarrow\!\!\$\ \Pi.\mathtt{Setup}()$

Setup(lid):

1 $\mathtt{st} \leftarrow \{\mathsf{epoch} : 0,\ g_{\mathsf{epoch}} : \mathrm{H}(\mathsf{epoch}),\ \mathtt{lid} : \mathtt{lid}\}$; Return st

Process(st, tx):

1 Require ($\mathtt{tx.lid} = \mathtt{st.lid}$ and $\mathtt{tx}.u \notin \mathcal{U}$ and $\Pi.\mathrm{V}(\mathsf{crs}, (\mathtt{st}, \mathtt{tx}), \mathtt{tx}.\pi)$)

2 If ($\mathtt{tx.type} = $ Debit) then

3 $\quad D \leftarrow \mathtt{tx}.D^{\mathtt{tx.val}^{-1} \cdot \mathtt{tx.fp}}$; For pk $\in \mathtt{tx.pk}$ do $\mathbf{C}[\mathrm{pk}] \leftarrow \mathtt{tx}.\mathbf{C}[\mathrm{pk}]^{\mathtt{tx.val}^{-1} \cdot \mathtt{tx.fp}}$

4 Else If ($\mathtt{tx.type} = $ Credit) then

5 $\quad D \leftarrow \mathtt{tx}.D^{\mathtt{tx.fp}}$; For pk $\in \mathtt{tx.pk}$ do $\mathbf{C}[\mathrm{pk}] \leftarrow \mathtt{tx}.\mathbf{C}[\mathrm{pk}]^{\mathtt{tx.fp}}$

6 Else $\mathbf{C}[\mathrm{pk}] \leftarrow \mathtt{tx}.\mathbf{C}[\mathrm{pk}]$; $D \leftarrow \mathtt{tx}.D$

7 For pk $\in \mathtt{tx.pk}$ do $\mathtt{st.Pending}[\mathrm{pk}] \leftarrow \mathtt{st.Pending}[\mathrm{pk}] \circ (\mathbf{C}[\mathrm{pk}], D)$

8 $\mathsf{epoch}' \leftarrow \mathsf{newepoch}(\mathtt{st})$ // Increment epoch as specified via newepoch

9 If ($\mathsf{epoch}' \neq \mathsf{epoch}$) then

10 $\quad \mathtt{st.epoch} \leftarrow \mathsf{epoch}'$; $\mathcal{U} \leftarrow \emptyset$; $\mathtt{st}.g_{\mathsf{epoch}'} \leftarrow \mathrm{H}(\mathsf{epoch}')$

11 $\quad$ For pk $\in$ Pending do

12 $\quad\quad \mathtt{st.Acc}[\mathrm{pk}] \leftarrow \mathtt{st.Acc}[\mathrm{pk}] \circ \mathtt{st.Pending}[\mathrm{pk}]$; $\mathtt{st.Pending}[\mathrm{pk}] \leftarrow (1_{\mathbb{G}}, 1_{\mathbb{G}})$

13 Return st

KeyGen():

14 $\mathrm{sk} \leftarrow\!\!\$\ [p]$; $\mathrm{pk} \leftarrow g^{\mathrm{sk}}$; $\mathrm{pk}.\pi \leftarrow\!\!\$\ \Pi.\mathrm{P}(\mathsf{crs}, \mathrm{pk}, \mathrm{sk})$; Return pk

KeyVf(pk):

15 Return $\Pi.\mathrm{V}(\mathsf{crs}, \mathrm{pk}, \mathrm{pk}.\pi)$

Read(st, sk):

16 $(C, D) \leftarrow \mathtt{st.Acc}[g^{\mathrm{sk}}]$; $B \leftarrow C \cdot D^{-\mathrm{sk}}$

17 For $b \in [0, \mathsf{MAX}]$ do If $(g^b = B)$ then return b

CreateTx(st, sk, (pk′, amt), val, AD):

18 $\mathrm{pk} \leftarrow g^{\mathrm{sk}}$; $b \leftarrow \mathtt{Read}(\mathtt{st}, \mathrm{sk})$

19 Require ($\mathrm{pk} \in \mathtt{AD.pk}$) ; Require ($\mathrm{pk}' = \perp \vee \mathrm{pk}' \in \mathtt{AD.pk}$)

20 For $y \in \mathtt{AD.pk}$ do $\mathbf{val}[y] \leftarrow 0$

21 $\mathbf{val}[\mathrm{pk}] \leftarrow -\mathtt{amt} + \mathtt{val}$; If $(\mathrm{pk}' \neq \perp)$ then $\mathbf{val}[\mathrm{pk}'] \leftarrow \mathtt{amt}$

22 $r \leftarrow\!\!\$\ \mathbb{Z}_p$; $\mathtt{tx} \leftarrow \{\mathtt{AD} : \mathtt{AD},\ D : g^r,\ u : (\mathtt{st}.g_{\mathsf{epoch}})^{\mathrm{sk}}\}$

23 For $y \in \mathtt{tx.pk}$ do $\mathtt{tx}.\mathbf{C}[y] \leftarrow y^r \cdot g^{\mathbf{val}[y]}$

24 $\mathtt{tx}.\pi \leftarrow \Pi.\mathrm{P}((\mathtt{st}, \mathtt{tx}), (\mathrm{sk}, r, b - \mathtt{amt} + \mathtt{val}, \mathtt{amt}, \mathrm{pk}, \mathrm{pk}'))$; Return tx

Figure 7: Construction of $\text{FLAX}_{\text{Zether}}$ system. The parameter of a FLAX transaction is stored as tx.fp. Function newepoch, whose specification is not shown here, increments the epoch as needed. The exact implementation of newepoch effects only the correctness of the system.

the form $(g, g_{\mathsf{epoch}}, \mathsf{Acc}', \mathtt{tx})$, or $(g, \mathsf{Acc}', \mathtt{tx})$, where $\mathtt{tx}$ is a key-value map specifying $\mathtt{tx.type}$, $\mathtt{tx}.u$ (only in case of transfer and debit transactions), $\mathtt{tx.val}$, $\mathtt{tx}.D$, $\mathtt{tx}.\mathbf{C}$, and $\mathtt{tx.AD}$. We note that these statements are all properties of either the ledger state $\mathtt{st}$ or transaction $\mathtt{tx}$. Hence, for notational convenience, we simply write $(\mathtt{st}, \mathtt{tx})$ to denote an instance of R. Formally, we also need to include the discrete-log relation R_{DL} into R to facilitate proofs of secret-key possessions. However, such usage of R is only required in $\mathtt{KeyGen}$, $\mathtt{KeyVf}$, and a specific step in the transaction privacy proof.

Instantiating FLAX. Let Π be a non-interactive proof system for relation R. Let H be a hash function modeled as a random oracle modeled as a random oracle with output space $\mathbb{G}$. The constituent algorithms of the transaction system are given in Figure 7. We require associated data to contain a set **pk**, denoting the selected *anonymity set*. We require that the public key of the originator g^{sk} to be in **pk**, and that the receiver public key $\mathrm{pk}' \neq \perp$ to also be in **pk**. Note that since the anonymity set is included in the associated data, it is accessible via $\mathtt{tx}.\mathbf{pk}$ for the constructed transaction $\mathtt{tx}$.

Epoch and correctness. We refer to [BAZB20] for more discussion on the table $\mathtt{Pending}$ and the use of epochs. Intuitively, this work-around is needed to ensure that each account owner can submit a transaction in any epoch. However, due to this, account balances are not usable during the same epoch and the construction only achieves a weaker form of correctness, where balances are guaranteed to be spendable *by the next epoch*.

Anonymity set. Similar to Zether, privacy for transactions come from the use of decoy public keys. Our syntax assumes that the anonymity set **pk** to be supplied in the associated data, meaning our model does not formally specify how such sets are picked. We give observations regarding the selection of anonymity sets *distinct* in our setting of composable DeFi, but leave the formal study and analysis required for future work. Suppose a Zether-based FLAX is deployed inside a token standard outline in Section 3. First, we remark that all FLAX transactions included in the same blockchain transaction should use the same anonymity set. For example, suppose a blockchain transaction that includes two FLAX transaction $\mathtt{tx}_1, \mathtt{tx}_2$. It is easy to deduce that the originator of the transaction must be inside the set $\mathtt{tx}_1.\mathbf{pk} \cap \mathtt{tx}_2.\mathbf{pk}$. Intuitively, the overall anonymity set is the intersection of all constituent ones. Second, the standard restriction in picking anonymity sets should be applied to all FLAX transactions for a single smart contract call. For example, consider an $\mathtt{EnterLP}$ call to a pool, which expects two debit transactions $\mathtt{dtx}_A$ and $\mathtt{dtx}_B$. The anonymity set of this blockchain transaction should only include those public keys that plausibly contain enough balance for both debit transactions. In upshot, selection of the anonymity set should be delegated to the wallet or client application for the smart contract platform.

Condensing Zether to a simple cryptographic primitive. Zether includes many design choices tailored to the functionality of the Ethereum blockchain in 2019. It is designed where the entire system, consisting of the Zether smart contract (ZSC) plus user algorithms, achieves what is known as a Decentralized Anonymous Payment (DAP) system. We take a more modular approach and acknowledge the distinction between cryptographic primitives and smart contracts. Cryptographic primitives, such as public-key encryption (PKE) and FLAX, must satisfy their associated security guarantees. Smart contracts can invoke cryptographic primitives to achieve additional functionality.

Features dropped from Zether. (1) lock and unlock: We drop the functionality of lock and unlock from Zether, since these features (of a smart contract) can be implemented exclusively using a signature scheme such as Schnorr or ECDSA. (2) Per account roll-over: Zether rolls the pending transaction state into the account state only when needed, due to high gas usage if this is done for all pending transactions. This amortization is not relevant in achieving any security notions, hence it is dropped in our treatment. (3) Mint and burn: these smart contract interfaces

can be realized, in an anonymous manner, via our debit and credit transactions instead.

Features added. First, FLAX allows debit and credit transactions which can be applied flexibly at processing time (i.e. execution time of the smart contract). Second, we explicitly added the "associated data" field to a transaction. Intuitively, the functionality of lock and unlock is moved to the associated data.

Conditional disclosure. For regulatory compliance, a user inside the anonymity set of a transaction might need to prove or disprove that it is the originator of a transaction. We note that a transaction for epoch with nonce $\mathtt{tx}.u$ is a originated from pk if and only if $(g, \mathrm{pk}, g_{\mathsf{epoch}}, \mathtt{tx}.u)$ is a valid DDH tuple. Hence, if a user needs to claim or disclaim a transaction, a proof of the correct nonce for the user at the specified epoch can be supplied.

6.2.3 Security

We verify consistency, transaction integrity, and transaction privacy for $\textsc{Flax}_{\text{Zether}}$. We show that $\textsc{Flax}_{\text{Zether}}$ is consistent as long as the employed proof system Π is sound. This follows by inspection of the relation R since all valid statements correctly declare their net value change. Hence, breaking consistency implies breaking soundness.

Theorem 6.1 $\textsc{Flax}_{\text{Zether}}$ *is consistent as long as* Π *is sound. Specifically, given any adversary* $\mathcal{A}_{\text{cons}}$*, the proof constructs adversaries* $\mathcal{A}_{\text{snd}}$ *such that*

$$\mathbf{Adv}^{\text{cons}}_{\textsc{Flax}_{\text{Zether}}}(\mathcal{A}_{\text{snd}}) \leq \mathbf{Adv}^{\text{snd}}_{\Pi}(\mathcal{A}_{\text{snd}}) \; . \tag{6}$$

For each $\mathtt{tx}$ *in the output of* $\mathcal{A}_{\text{cons}}$*, snd-adversary* $\mathcal{A}_{\text{snd}}$ *makes a query to* $\textsc{Vf}$*.*

Next, we show transaction integrity. We give the following theorem.

Theorem 6.2 *Suppose that DL and DDH problems are hard in group* $\mathbb{G}$ *with generator* g *and that* Π *is zero-knowledge and simulation-extractable. Then,* $\textsc{Flax}_{\text{Zether}}$ *satisfies transaction integrity. Specifically, given any adversary* $\mathcal{A}_{\text{tx-int}}$*, the proof constructs adversaries* $\mathcal{A}_{\text{zk}}$*,* $\mathcal{A}_{\text{xt}}$*,* $\mathcal{A}_{\text{ddh}}$*, and* $\mathcal{A}_{\text{dl}}$*, all as efficient as* $\mathcal{A}_{\text{tx-int}}$*, such that*

$$\begin{aligned} \mathbf{Adv}^{\text{tx-int}}_{\textsc{Flax}_{\text{Zether}}}(\mathcal{A}_{\text{tx-int}}) \leq \; & \mathbf{Adv}^{\text{zk}}_{\Pi}(\mathcal{A}_{\text{zk}}) + \mathbf{Adv}^{\text{xt}}_{\Pi}(\mathcal{A}_{\text{xt}}) \\ & + \mathbf{Adv}^{\text{ddh}}_{\mathbb{G},g,q}(\mathcal{A}_{\text{ddh}}) + \mathbf{Adv}^{\text{dl}}_{\mathbb{G},g}(\mathcal{A}_{\text{dl}}) \; , \end{aligned} \tag{7}$$

where q *(parameter to q-DDH) is the maximum amount of queries to* $\textsc{CreateTx}$ *oracle that* $\mathcal{A}_{\text{tx-int}}$ *makes.*

The proof is straightforward with the right ingredients. We will use the properties in the following order in constructing hybrid experiments: zero-knowledge, DDH, simulation extractability, then finally DL. Zero-knowledge is first used to simulate proofs for honestly generated transactions without using the target secret key sk. Next, we move to a game where the transaction nonces are sampled uniformly randomly, instead of computed as $g^{\text{sk}}_{\mathsf{epoch}}$. After these two changes, transactions can be simulated without knowledge of target sk, hence we can use XT to bound the probability that the final forged transaction is both valid and decreases the balance of the target public key pk, but does not lead to a valid extraction. Finally, in the last hybrid game, valid extraction of sk is as hard as solving DL.

Proof of Theorem 6.2: Consider the game sequence given in Figure 8.

Moving from game $\mathbf{G}_0$ to $\mathbf{G}_1$, we have switched the crs from real to simulated, as well as changed all proof generation to use the simulator $\Pi.\mathsf{S}$. The distance between them is upper bounded by

Game $\mathbf{G}_0$-$\mathbf{G}_3$

$\underline{\mathbf{G}_0}$: $\mathsf{crs} \leftarrow\!\!\$\ \Pi.\mathtt{Setup}$
$\underline{\mathbf{G}_1\text{-}\mathbf{G}_3}$: $(\mathsf{crs}, \mathsf{td}) \leftarrow\!\!\$\ \Pi.\mathsf{SSetup}$
$\mathrm{sk} \leftarrow\!\!\$\ [p]$; $\mathrm{pk} \leftarrow g^{\mathrm{sk}}$; $(\mathtt{st}^*, \mathtt{tx}^*) \leftarrow \mathcal{A}^{\textsc{CreateTx}}_{\textsc{Flax}}(\mathsf{crs}, \mathrm{pk})$
Require $(\mathtt{tx}^* \notin S)$; $\mathtt{st}' \leftarrow \mathtt{Process}(\mathtt{st}^*, \mathtt{tx}^*)$
$b \leftarrow (\mathtt{Read}(\mathtt{st}', \mathrm{sk}) < \mathtt{Read}(\mathtt{st}^*, \mathrm{sk}))$
$\underline{\mathbf{G}_0\text{-}\mathbf{G}_2}$: Return b
$\underline{\mathbf{G}_3}$: Return $(b \wedge (\mathrm{sk} \in \Pi.\mathsf{Ext}(\mathsf{crs}, \mathsf{td}, (\mathtt{st}^*, \mathtt{tx}^*), \mathtt{tx}.\pi)))$

$\textsc{CreateTx}(\mathtt{st}, (\mathrm{pk}', \mathtt{amt}), \mathtt{val}, \mathtt{AD})$:
$\mathrm{pk} \leftarrow g^{\mathrm{sk}}$; $b \leftarrow \mathtt{Read}(\mathtt{st}, \mathrm{sk})$
For $y \in \mathtt{AD}.\mathbf{pk}$ do $\mathbf{val}[y] \leftarrow 0$
$\mathbf{val}[\mathrm{pk}] \leftarrow -\mathtt{amt} + \mathtt{val}$; If $(\mathrm{pk}' \neq \perp)$ then $\mathbf{val}[\mathrm{pk}'] \leftarrow \mathtt{amt}$
$r \leftarrow\!\!\$\ \mathbb{Z}_p$; $\mathtt{tx} \leftarrow \{\mathtt{AD} : \mathtt{AD},\ D : g^r\}$
For $y \in \mathtt{tx}.\mathbf{pk}$ do $\mathtt{tx}.\mathbf{C}[y] \leftarrow y^r \cdot g^{\mathbf{val}[y]}$
$\underline{\mathbf{G}_0,\ \mathbf{G}_1}$: $\mathtt{tx}.u \leftarrow (\mathtt{st}.g_{\mathsf{epoch}})^{\mathrm{sk}}$
$\underline{\mathbf{G}_2,\ \mathbf{G}_3}$: $\mathtt{tx}.u \leftarrow\!\!\$\ \mathbb{G}$
$\underline{\mathbf{G}_0}$: $\mathtt{tx}.\pi \leftarrow \Pi.\mathrm{P}(\mathsf{crs}, (\mathtt{st}, \mathtt{tx}), (\mathrm{sk}, r, b - \mathtt{amt} + \mathtt{val}, \mathtt{amt}, \mathrm{pk}, \mathrm{pk}'))$
$\underline{\mathbf{G}_1\text{-}\mathbf{G}_3}$: $\mathtt{tx}.\pi \leftarrow \Pi.\mathsf{S}(\mathsf{crs}, (\mathtt{st}, \mathtt{tx}), \mathsf{td})$
$S \xleftarrow{\cup} \mathtt{tx}$; Return $\mathtt{tx}$

Figure 8: Game sequence for the proof of Theorem 6.2.

the ZK advantage of a reduction adversary that is as efficient as $\mathcal{A}_{\textsc{Flax}}$. Hence, we could construct adversary $\mathcal{A}_{\mathrm{zk}}$ such that

$$\Pr[\mathbf{G}_0] - \Pr[\mathbf{G}_1] \leq \mathbf{Adv}^{\mathrm{zk}}_{\Pi}(\mathcal{A}_{\mathrm{zk}}) \ . \tag{8}$$

Moving from game $\mathbf{G}_1$ to $\mathbf{G}_2$, we have are switching the transaction nonces to be uniform randomly sampled from the group $\mathbb{G}$. It is routine to check that the distance is upper bounded by the q-DDH advantage of an efficient reduction adversary. Hence, we can construct $\mathcal{A}_{\mathrm{ddh}}$ such that

$$\Pr[\mathbf{G}_1] - \Pr[\mathbf{G}_2] \leq \mathbf{Adv}^{\mathrm{ddh}}_{\mathbb{G},g,q}(\mathcal{A}_{\mathrm{ddh}}) \ . \tag{9}$$

Note that, in game $\mathbf{G}_2$, transaction creation no longer needs the secret key sk. We would like to run the extraction $\Pi.\mathsf{Ext}$ on the forged transaction $\mathtt{tx}^*$ to extract sk. Consider game $\mathbf{G}_3$, which additionally check that the extraction of $(\mathtt{st}^*, \mathtt{tx}^*)$ yields a witness containing target secret key sk. We claim that $\mathbf{G}_2$ and $\mathbf{G}_3$ should be close unless simulation extractability is violated. This is because, if b_1 is true, then the statement $(\mathtt{st}^*, \mathtt{tx}^*)$ is fresh and valid. This means that XT guarantees a valid extraction. Hence, we could give adversary $\mathcal{A}_{\mathrm{xt}}$ such that

$$\Pr[\mathbf{G}_2] - \Pr[\mathbf{G}_3] \leq \mathbf{Adv}^{\mathrm{xt}}_{\Pi}(\mathcal{A}_{\mathrm{xt}}) \ . \tag{10}$$

Finally, we note that an adversary winning game $\mathbf{G}_3$ can be turned into a DL adversary, since sk is not used anywhere inside $\mathbf{G}_3$, only the target key pk. Hence, we could give DL adversary $\mathcal{A}_{\mathrm{dl}}$ such that

$$\Pr[\mathbf{G}_3] \leq \mathbf{Adv}^{\mathrm{dl}}_{\mathbb{G},g}(\mathcal{A}_{\mathrm{dl}}) \ . \tag{11}$$

Note that all constructed adversaries run $\mathcal{A}_{\textsc{Flax}}$ as a black-box and involve simulation overhead of the games, which is small. Putting the above equations together we conclude the proof. ∎

Next, we show that $\textsc{Flax}_{\textsc{Zether}}$ satisfies replay protection as long as Π is sound. If Π is sound

then $(g, g_{\mathsf{epoch}}, \mathrm{pk}, \mathtt{tx}.u)$ must be a DDH-tuple for a valid transaction $\mathtt{tx}$, where pk is the public key whose balance decreases in $\mathtt{tx}$. This means that there is exactly one transaction nonce u that is considered valid for each account during an epoch. Moreover, as long as g_{epoch} does not repeat, the correct transaction nonces do not repeat either. Hence, we observe the following theorem.

Theorem 6.3 $\textsc{Flax}_{\mathrm{Zether}}$ *satisfies replay protection if* Π *is sound. Formally, given adversary* $\mathcal{A}_{\mathrm{rep}}$*, adversary* $\mathcal{A}_{\mathrm{snd}}$ *can be constructed so that*

$$\mathbf{Adv}^{\mathrm{rep}}_{\mathrm{FLAX}_{\mathrm{Zether}}}(\mathcal{A}_{\mathrm{rep}}) \leq \mathbf{Adv}^{\mathrm{snd}}_{\Pi}(\mathcal{A}_{\mathrm{snd}}) + \frac{q^2}{p} \ . \tag{12}$$

where q *is the number of queries that* $\mathcal{A}_{\mathrm{rep}}$ *makes to* $\textsc{ProcessO}$*.*

Finally, we show that $\textsc{Flax}_{\mathrm{Zether}}$ satisfies transaction privacy. First, we define $\mathsf{Anonymize}$.

$\mathsf{Anonymize}(\mathtt{st}, \mathbf{Keys}, \mathrm{pk}_0, (\mathrm{pk}'_0, \mathtt{amt}), \mathtt{val}, \mathtt{AD})$:
1 $\mathbf{R} \leftarrow \mathbf{Keys}.\mathtt{keys} \cap \mathtt{AD}.\mathbf{val}$
2 Require $(\mathrm{pk}_0 \in \mathbf{R})$
3 $\mathrm{pk}_1 \leftarrow\!\!\$ \{\mathrm{pk} \in \mathbf{R} \mid \mathtt{Read}(\mathtt{st}, \mathbf{Keys}[\mathrm{pk}]) \geq \mathtt{amt} - \mathtt{val}\}$
4 If $(\mathrm{pk}'_0 \in \mathbf{R})$ then
5 $\quad \mathrm{pk}'_1 \leftarrow\!\!\$ \mathbf{R}$; $\mathtt{amt}' \leftarrow\!\!\$ [\mathtt{Read}(\mathtt{st}, \mathbf{Keys}[\mathrm{pk}_1]) + \mathtt{val}]$
6 Else $(\mathrm{pk}'_1, \mathtt{amt}') \leftarrow (\mathrm{pk}'_0, \mathtt{amt})$
7 Return $(\mathrm{pk}_1, \mathrm{pk}'_1, \mathtt{amt}')$

Theorem 6.4 *Suppose that DDH problem is hard in group* $\mathbb{G}$ *with generator* g*, and that* Π *is zero-knowledge and simulation-extractable. Then,* $\textsc{Flax}_{\mathrm{Zether}}$ *satisfies transaction privacy. Specifically, given any adversary* $\mathcal{A}_{\text{tx-priv}}$*, the proof constructs adversaries* $\mathcal{A}_{\mathrm{zk}}$*,* $\mathcal{A}_{\mathrm{xt}}$*, and* $\mathcal{A}_{\mathrm{ddh}}$*, all as efficient as* $\mathcal{A}_{\text{tx-int}}$*, such that*

$$\begin{aligned}\mathbf{Adv}^{\text{tx-int}}_{\mathrm{FLAX}_{\mathrm{Zether}},\mathsf{Anonymize}}(\mathcal{A}_{\text{tx-priv}}) \leq\ & \mathbf{Adv}^{\mathrm{zk}}_{\Pi}(\mathcal{A}_{\mathrm{zk}}) + \mathbf{Adv}^{\mathrm{xt}}_{\Pi}(\mathcal{A}_{\mathrm{xt}}) \\ & + \mathbf{Adv}^{\mathrm{ddh}}_{\mathbb{G},g,r}(\mathcal{A}_{\mathrm{ddh}}) \ ,\end{aligned} \tag{13}$$

where r *is the size of the anonymity ring.*

Proof of Theorem 6.4: Consider the game sequence given in Figure 9. The first game $\mathbf{G}_0$ is the tx-priv game associated with $\textsc{Flax}_{\mathrm{Zether}}$ and $\mathsf{Anonymize}$. We switch the proof generation from real to simulated in game $\mathbf{G}_1$. It is routine to construct zk-adversary $\mathcal{A}_{\mathrm{zk}}$ such that

$$\Pr[\mathbf{G}_0] - \Pr[\mathbf{G}_1] \leq \mathbf{Adv}^{\mathrm{zk}}_{\mathrm{FLAX}_{\mathrm{Zether}}}(\mathcal{A}_{\mathrm{zk}}) \ . \tag{14}$$

Next, we would like to extract out secret keys from the adversary. This is exactly the code addition in game $\mathbf{G}_2$, which extracts the secret key in the ring that is not honest (line 16). It is routine to give an xt-adversary, such that

$$\Pr[\mathbf{G}_1] - \Pr[\mathbf{G}_2] \leq \mathbf{Adv}^{\mathrm{xt}}_{\Pi}(\mathcal{A}_{\mathrm{xt}}) \ . \tag{15}$$

We are now ready to switch group elements to uniformly random using DDH. First is the transaction nonce $\mathtt{tx}.u$. Next are the ciphertexts $\mathtt{tx}.\mathbf{C}[y]$ for those keys y that are (1) honest and (2) have balances at least $\mathtt{amt} - \mathtt{val}$. We claim that

$$\Pr[\mathbf{G}_2] - \Pr[\mathbf{G}_3] \leq \mathbf{Adv}^{\mathrm{ddh}}_{\mathbb{G},g,r}(\mathcal{A}_{\mathrm{ddh}}) \ , \tag{16}$$

where r is upper bounded by the size of the anonymity ring. Finally, we claim

$$\Pr[\mathbf{G}_3] = \frac{1}{2} \ . \tag{17}$$

Combining the above equations, we obtain (13).∎

Game $\mathbf{G}_0$-$\mathbf{G}_3$

$\underline{\mathbf{G}_0}$: $\mathsf{crs} \leftarrow\!\!\$\ \Pi.\mathtt{Setup}$
$\underline{\mathbf{G}_1\text{-}\mathbf{G}_3}$: $(\mathsf{crs}, \mathsf{td}) \leftarrow\!\!\$\ \Pi.\mathsf{SSetup}$
$(\mathtt{st}, \mathrm{pk}_0, (\mathrm{pk}'_0, \mathtt{amt}), \mathtt{val}, \mathtt{AD}) \leftarrow \mathcal{A}^{\textsc{NewUser},\textsc{CreateTx}}_{\textsc{Flax}}(\mathsf{crs})$
Require $(\mathrm{pk}_0 \in \mathbf{Keys})$
$(\mathrm{pk}_1, \mathrm{pk}'_1, \mathtt{amt}'_1) \leftarrow\!\!\$\ \mathsf{Anonymize}(\mathtt{st}, \mathbf{Keys}, \mathrm{pk}_0, (\mathrm{pk}'_0, \mathtt{amt}_0), \mathtt{val}, \mathtt{AD})$
$\mathtt{tx} \leftarrow\!\!\$\ \mathtt{CreateTx}(\mathtt{st}, \mathbf{Keys}[\mathrm{pk}_b], (\mathrm{pk}'_b, \mathtt{amt}_b), \mathtt{val}, \mathtt{AD})$
$b' \leftarrow\!\!\$\ \mathcal{A}(\mathtt{tx})$; Return $(b = b')$

NewUser:
$(\mathrm{pk}, \mathrm{sk}) \leftarrow\!\!\$\ \mathtt{KeyGen}()$; $\mathbf{Keys}[\mathrm{pk}] \leftarrow \mathrm{sk}$; Return pk

CreateTx$(\mathtt{st}, \mathrm{pk}, (\mathrm{pk}', \mathtt{amt}), \mathtt{val}, \mathtt{AD})$:
For $y \in \mathtt{AD}.\mathbf{pk}$ do $\mathbf{val}[y] \leftarrow 0$
$\mathbf{val}[\mathrm{pk}] \leftarrow -\mathtt{amt} + \mathtt{val}$; If $(\mathrm{pk}' \neq \bot)$ then $\mathbf{val}[\mathrm{pk}'] \leftarrow \mathtt{amt}$
$r \leftarrow\!\!\$\ \mathbb{Z}_p$; $\mathtt{tx} \leftarrow \{\mathtt{AD} : \mathtt{AD},\ D : g^r\}$
For $y \in \mathtt{tx}.\mathbf{pk}$ do $\mathtt{tx}.\mathbf{C}[y] \leftarrow y^r \cdot g^{\mathbf{val}[y]}$
$\mathtt{tx}.u \leftarrow (\mathtt{st}.g_{\mathsf{epoch}})^{\mathbf{Keys}[\mathrm{pk}]}$
$\underline{\mathbf{G}_2, \mathbf{G}_3}$:
  For $y \in (\mathtt{AD}.\mathrm{pk} - \mathbf{Keys}.\mathtt{keys})$ do
    $\mathbf{Keys}[y] \leftarrow \Pi.\mathsf{Ext}(\mathsf{crs}, \mathsf{td}, y, y.\pi)$
    Require $(g^{\mathbf{Keys}[y]} = y)$
$\underline{\mathbf{G}_3}$:
  $\mathtt{tx}.u \leftarrow\!\!\$\ \mathbb{G}$
  For $y \in \mathtt{tx}.\mathbf{pk} \cap \mathbf{Keys}.\mathtt{keys}$ do
    If $(\mathtt{Read}(\mathtt{st}, \mathbf{Keys}[y]) \geq \mathtt{amt} - \mathtt{val})$ then $\mathtt{tx}.\mathbf{C}[y] \leftarrow\!\!\$\ \mathbb{G}$
$\underline{\mathbf{G}_0}$:
  $\mathrm{sk} \leftarrow \mathbf{Keys}[\mathrm{pk}]$; $b \leftarrow \mathtt{Read}(\mathtt{st}, \mathrm{sk})$
  $\mathtt{tx}.\pi \leftarrow \Pi.\mathrm{P}(\mathsf{crs}, (\mathtt{st}, \mathtt{tx}), (\mathrm{sk}, r, b - \mathtt{amt} + \mathtt{val}, \mathtt{amt}, \mathrm{pk}, \mathrm{pk}'))$
$\underline{\mathbf{G}_1\text{-}\mathbf{G}_3}$: $\mathtt{tx}.\pi \leftarrow \Pi.\mathsf{S}(\mathsf{crs}, (\mathtt{st}, \mathtt{tx}), \mathsf{td})$
Return $\mathtt{tx}$

Figure 9: Game sequence for the proof of Theorem 6.4.

Acknowledgements

We thank Tatsuaki Okamoto and Go Yamamoto for their helpful discussions during the early stages of this work. We thank Psi Vesely for their helpful editorial comments.

References

[Aav] Aave. Protocol Whitepaper. `https://github.com/aave/protocol-v2/raw/master/aave-v2-whitepaper.pdf`. Accessed Sept. 2021. 3, 5, 13, 16

[AC20] Guillermo Angeris and Tarun Chitra. Improved price oracles: Constant function market makers. In *Proceedings of the 2nd ACM Conference on Advances in Financial Technologies*, pages 80–91, 2020. 14

[AEC21] Guillermo Angeris, Alex Evans, and Tarun Chitra. A note on privacy in constant function market makers. *arXiv preprint arXiv:2103.01193*, 2021. 3

[AEF+21] Hendrik Amler, Lisa Eckey, Sebastian Faust, Marcel Kaiser, Philipp Sandner, and Benjamin Schlosser. DeFi-ning DeFi: Challenges & Pathway, 2021. 3

[AKC+19] Guillermo Angeris, Hsien-Tang Kao, Rei Chiang, Charlie Noyes, and Tarun Chitra. An analysis of uniswap markets. *arXiv preprint arXiv:1911.03380*, 2019. 14

[AZR] Hayden Adams, Noah Zinsmeister, and Dan Robinson. Uniswap v2 Core. `https://uniswap.org/whitepaper.pdf`. Accessed Sept. 2021. 3, 5, 13

[BAZB20] Benedikt Bünz, Shashank Agrawal, Mahdi Zamani, and Dan Boneh. Zether: Towards privacy in a smart contract world. In Joseph Bonneau and Nadia Heninger, editors, *FC 2020*, volume 12059 of *LNCS*, pages 423–443. Springer, Heidelberg, February 2020. 3, 6, 8, 18, 20, 24

[BBB+18] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. Bulletproofs: Short proofs for confidential transactions and more. In *2018 IEEE Symposium on Security and Privacy*, pages 315–334. IEEE Computer Society Press, May 2018. 8, 19

[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 *2014 IEEE Symposium on Security and Privacy*, pages 459–474. IEEE Computer Society Press, May 2014. 3, 6, 8, 18, 19

[BCG+20] Sean Bowe, Alessandro Chiesa, Matthew Green, Ian Miers, Pratyush Mishra, and Howard Wu. ZEXE: Enabling decentralized private computation. In *2020 IEEE Symposium on Security and Privacy*, pages 947–964. IEEE Computer Society Press, May 2020. 4

[Bel20] Mihir Bellare. Lectures on NIZKs. 2020. `https://cseweb.ucsd.edu/~mihir/cse208-Wi20/main.pdf`. 20

[BKSV21] Karim Baghery, Markulf Kohlweiss, Janno Siim, and Mikhail Volkhov. Another look at extraction and randomization of groth's zk-snark. *FC 2021*, 2021. 8, 19

[BR06] Mihir Bellare and Phillip Rogaway. The security of triple encryption and a framework for code-based game-playing proofs. In Serge Vaudenay, editor, *EUROCRYPT 2006*, volume 4004 of *LNCS*, pages 409–426. Springer, Heidelberg, May / June 2006. 16

[CXZ21] Shumo Chu, Yu Xia, and Zhenfei Zhang. Manta: a plug and play private defi stack. Cryptology ePrint Archive, Report 2021/743, 2021. `https://ia.cr/2021/743`. 4

[CZK+19] Raymond Cheng, Fan Zhang, Jernej Kos, Warren He, Nicholas Hynes, Noah Johnson, Ari Juels, Andrew Miller, and Dawn Song. Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contracts. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 185–200. IEEE, 2019. 4

[DDO+01] Alfredo De Santis, Giovanni Di Crescenzo, Rafail Ostrovsky, Giuseppe Persiano, and Amit Sahai. Robust non-interactive zero knowledge. In Joe Kilian, editor,

CRYPTO 2001, volume 2139 of *LNCS*, pages 566–598. Springer, Heidelberg, August 2001. 8, 19, 21

[DGK+20] Philip Daian, Steven Goldfeder, Tyler Kell, Yunqi Li, Xueyuan Zhao, Iddo Bentov, Lorenz Breidenbach, and Ari Juels. Flash boys 2.0: Frontrunning in decentralized exchanges, miner extractable value, and consensus instability. In *2020 IEEE Symposium on Security and Privacy*, pages 910–927. IEEE Computer Society Press, May 2020. 3, 7

[Dia20] Benjamin E. Diamond. Many-out-of-many proofs and applications to anonymous zether. Cryptology ePrint Archive, Report 2020/293, 2020. `https://eprint.iacr.org/2020/293`. 3, 6, 8, 18, 20

[Ego] Michael Egorov. StableSwap—efficient mechanism for Stablecoin liquidity. `https://curve.fi/files/stableswap-paper.pdf`. Accessed Sept. 2021. 3, 5, 13

[EMP+21] Felix Engelmann, Lukas Müller, Andreas Peter, Frank Kargl, and Christoph Bösch. SwapCT: Swap confidential transactions for privacy-preserving multi-token exchanges. *PoPETs*, 2021(4):270–290, October 2021. 4

[Eth] Etherscan. Token tracker. `https://etherscan.io/tokens`. Accessed Sept. 2021. 3

[FKMV12] Sebastian Faust, Markulf Kohlweiss, Giorgia Azzurra Marson, and Daniele Venturi. On the non-malleability of the Fiat-Shamir transform. In Steven D. Galbraith and Mridul Nandi, editors, *INDOCRYPT 2012*, volume 7668 of *LNCS*, pages 60–79. Springer, Heidelberg, December 2012. 19

[FMMO19] Prastudy Fauzi, Sarah Meiklejohn, Rebekah Mercer, and Claudio Orlandi. Quisquis: A new design for anonymous cryptocurrencies. In Steven D. Galbraith and Shiho Moriai, editors, *ASIACRYPT 2019, Part I*, volume 11921 of *LNCS*, pages 649–678. Springer, Heidelberg, December 2019. 3, 6, 8, 18, 19

[FOS19] Georg Fuchsbauer, Michele Orrù, and Yannick Seurin. Aggregate cash systems: A cryptographic investigation of Mimblewimble. In Yuval Ishai and Vincent Rijmen, editors, *EUROCRYPT 2019, Part I*, volume 11476 of *LNCS*, pages 657–689. Springer, Heidelberg, May 2019. 3, 6, 18, 20

[GM17] Jens Groth and Mary Maller. Snarky signatures: Minimal signatures of knowledge from simulation-extractable SNARKs. In Jonathan Katz and Hovav Shacham, editors, *CRYPTO 2017, Part II*, volume 10402 of *LNCS*, pages 581–612. Springer, Heidelberg, August 2017. 8, 19

[Gro06] Jens Groth. Simulation-sound NIZK proofs for a practical language and constant size group signatures. In Xuejia Lai and Kefei Chen, editors, *ASIACRYPT 2006*, volume 4284 of *LNCS*, pages 444–459. Springer, Heidelberg, December 2006. 8, 19, 21

[Gro16] Jens Groth. On the size of pairing-based non-interactive arguments. In Marc Fischlin and Jean-Sébastien Coron, editors, *EUROCRYPT 2016, Part II*, volume 9666 of *LNCS*, pages 305–326. Springer, Heidelberg, May 2016. 8, 19

[Jed] Tom Elvis Jedusor. Mimble Wimble. `https://download.wpsoftware.net/bitcoin/wizardry/mimblewimble.txt`. 3, 20

[KGC+18] Harry A. Kalodner, Steven Goldfeder, Xiaoqi Chen, S. Matthew Weinberg, and Edward W. Felten. Arbitrum: Scalable, private smart contracts. In William Enck and Adrienne Porter Felt, editors, *USENIX Security 2018*, pages 1353–1370. USENIX Association, August 2018. 4

[KKK21] Thomas Kerber, Aggelos Kiayias, and Markulf Kohlweiss. Kachina–foundations of private smart contracts. In *2021 IEEE 34th Computer Security Foundations Symposium (CSF)*, pages 1–16. IEEE, 2021. 4

[KLO20] Jihye Kim, Jiwon Lee, and Hyunok Oh. Simulation-extractable zk-snark with a single verification. *IEEE Access*, 8:156569–156581, 2020. 8, 19

[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 *2016 IEEE Symposium on Security and Privacy*, pages 839–858. IEEE Computer Society Press, May 2016. 4

[KZM+15] Ahmed Kosba, Zhichao Zhao, Andrew Miller, Yi Qian, Hubert Chan, Charalampos Papamanthou, Rafael Pass, abhi shelat, and Elaine Shi. How to use SNARKs in universally composable protocols. Cryptology ePrint Archive, Report 2015/1093, 2015. `https://eprint.iacr.org/2015/1093`. 19

[LCO+16] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 254–269. ACM Press, October 2016. 3, 9, 10

[LH] Robert Leshner and Geoffrey Hayes. Compound: The Money Market Protocol. `https://compound.finance/documents/Compound.Whitepaper.pdf`. Accessed Sept. 2021. 3, 5, 13, 16

[Max] Gregory Maxwell. CoinJoin Bitcoin privacy for the real world. `https://bitcointalk.org/index.php?topic=279249.0`. Accessed Nov. 2021. 4

[MM18] Sarah Meiklejohn and Rebekah Mercer. Möbius: Trustless tumbling for transaction privacy. *PoPETs*, 2018(2):105–121, April 2018. 4

[MWD] Merriam-Webster.com Dictionary. Flax. `https://www.merriam-webster.com/dictionary/flax`. Accessed Sept. 2021. 5

[Noe15] Shen Noether. Ring signature confidential transactions for monero. Cryptology ePrint Archive, Report 2015/1098, 2015. `https://eprint.iacr.org/2015/1098`. 3, 8, 19

[Poe] Andrew Poelstra. Mimble Wimble. `https://download.wpsoftware.net/bitcoin/wizardry/mimblewimble.pdf`. 3, 20

[PSS] Alexey Pertsev, Roman Semenov, and Roman Storm. Tornado Cash Privacy Solution. `https://berkeley-defi.github.io/assets/material/Tornado%20Cash%20Whitepaper.pdf`. 4

[Pul] DeFi Pulse. Defi Pulse: The Decentralized Finance Leaderboard. `https://defipulse.com`. Accessed Sept. 2021. 3

[QZLG20] Kaihua Qin, Liyi Zhou, Benjamin Livshits, and Arthur Gervais. Attacking the defi ecosystem with flash loans for fun and profit. *arXiv preprint arXiv:2003.03810*, 2020. 3

[Sah99] Amit Sahai. Non-malleable non-interactive zero knowledge and adaptive chosen-ciphertext security. In *40th FOCS*, pages 543–553. IEEE Computer Society Press, October 1999. 8, 19, 21

[SALY17] Shi-Feng Sun, Man Ho Au, Joseph K. Liu, and Tsz Hon Yuen. RingCT 2.0: A compact accumulator-based (linkable ring signature) protocol for blockchain cryptocurrency monero. In Simon N. Foley, Dieter Gollmann, and Einar Snekkenes, editors, *ESORICS 2017, Part II*, volume 10493 of *LNCS*, pages 456–474. Springer, Heidelberg, September 2017. 3, 8, 18, 19

[SBG+19] Samuel Steffen, Benjamin Bichsel, Mario Gersbach, Noa Melchior, Petar Tsankov, and Martin T. Vechev. zkay: Specifying and enforcing data privacy in smart contracts. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019*, pages 1759–1776. ACM Press, November 2019. 4

[Sch21] Fabian Schär. Decentralized finance: On blockchain-and smart contract-based financial markets. *Federal Reserve Bank of St. Louis Review*, Second Quarter 2021. 3

[Tea] The Maker Team. The Dai Stablecoin System. `https://makerdao.com/whitepaper/DaiDec17WP.pdf`. Accessed Sept. 2021. 3, 5, 13, 16

[VB] Fabian Vogelsteller and Vitalik Buterin. EIP-20: Token Standard. `https://eips.ethereum.org/EIPS/eip-20`. Accessed Sept. 2021. 3, 10

[WPG+21] Sam M Werner, Daniel Perez, Lewis Gudgeon, Ariah Klages-Mundt, Dominik Harz, and William J Knottenbelt. SoK: Decentralized Finance (DeFi). *arXiv preprint arXiv:2101.08778*, 2021. 3

[Yea] Yearn.finance. Introduction. `https://docs.yearn.finance/`. Accessed Sept. 2021. 3, 5, 13

[YSL+20] Tsz Hon Yuen, Shifeng Sun, Joseph K. Liu, Man Ho Au, Muhammed F. Esgin, Qingzhao Zhang, and Dawu Gu. RingCT 3.0 for blockchain confidential transaction: Shorter size and stronger security. In Joseph Bonneau and Nadia Heninger, editors, *FC 2020*, volume 12059 of *LNCS*, pages 464–483. Springer, Heidelberg, February 2020. 3, 6, 8, 18, 19