NPCI x Antaragni Hackathon

Chayan Kumawat 220309

Nikhil Mishra 210668

Ritvik Goyal 220898

October 18, 2024

Contents

Chapter 1

Introduction

Payment systems can be classified into two types: **register-based** and **value-based**.

- **Register-based**: In these systems, transactions are tracked by updating a central ledger, where details like sender, receiver, and amounts are logged. UPI (Unified Payments Interface) is an example, where each transfer is linked to user accounts and stored centrally.

- **Value-based**: These systems work like cash, where value is transferred without reliance on a central ledger. Transactions remain anonymous, and no personal details are logged, similar to cryptocurrencies like Bitcoin.

This book explains the design and implementation of a value-based payment system, discussing in-depth how to create a practical, privacy-preserving, and secure (micro-)payment protocol that integrates seamlessly with the modern web. Our value-based payment protocol can, in principle, operate on top of any existing register-based system.

1.1 Design Goals of the Solution

1. Privacy for Buyers

Our solution ensures that buyers' privacy is protected via **technical measures**, not just policies. Especially for **micropayments**, where disclosing private information such as personal details or spending habits could be disproportionate, our system guarantees that minimal data is collected. This allows for compliance with strict privacy regulations such as **GDPR**, while also preventing issues like **phishing** and **credit card fraud**, thus improving the overall security of the system.

2. Income Transparency for State Taxation

A major challenge for digital payment systems is **tax evasion** and **illegal business activities**. Our solution addresses this by allowing **income transparency**, which ensures that payments can be traced for **taxation purposes** while still maintaining user privacy for legitimate activities. This makes the system legal and **compliant with tax laws**, ensuring the state's ability to crack down on illegal transactions.

3. Prevention of Payment Fraud

Our solution implements **robust security measures** to prevent various forms of **payment fraud**. This includes cryptographic evidence for payment processing, eliminating the risks of **misleading interfaces** or fraudulent manipulation of the payment process. By emphasizing secure and transparent cryptographic processes, users and merchants are protected from fraud risks inherent in many existing payment systems.

4. Minimal Disclosure of Information

While maintaining transparency for necessary parties, our solution is designed to minimize information disclosure. This applies to both **buyers and merchants**, ensuring that sensitive details such as **financial information** remain hidden from competitors or third parties, except where legally necessary. This approach supports privacy while enabling necessary compliance with legal and financial regulations.

5. Usability for Non-experts

We focus on making the system **easy to use for non-expert customers and merchants**. The payment interface and integration are designed with simplicity in mind, ensuring **ease of use** without compromising security. Complex cryptographic operations are encapsulated within isolated components, making the system accessible even to those without technical expertise.

6. Operational Efficiency

In contrast to systems that rely on **resource-intensive proof-of-work** or similar mechanisms, our solution is built for **efficiency**. It can handle **micropayments** swiftly, making it suitable for environments where small transactions are common, such as content subscriptions or online services. This focus on efficiency ensures that the solution is viable in a wide range of scenarios.

7. Avoidance of Single Points of Failure

To ensure system reliability, our architecture is designed to **avoid single points of failure**. By isolating components and allowing independent audits, we reduce the risks associated with centralization. Even though the system architecture is relatively centralized, measures are in place to **distribute risks** and ensure high reliability.

8. Encouragement of Competition

Our solution fosters competition by reducing the barriers for new entrants into the payment ecosystem. By splitting the system into **modular components** that can be independently developed and integrated, we encourage **innovation** and **competition**. This approach prevents vendor lock-in, ensuring that the system remains open for further enhancements and alternative providers.

9. Income-transparent Change Mechanism

Our system introduces a unique approach to **change handling** in digital payments. Instead of simply dividing coins, which could lead to **double-spending issues**, we employ a method where remaining coin values can be used to **withdraw fresh, unlinkable coins**. This approach also maintains income transparency, preventing the misuse of change for **untaxed transactions**.

10. Prevention of Double Spending

To prevent double spending, all coins must be **immediately deposited online** during transactions. This real-time verification avoids the issues seen in other systems where offline merchants face delays in detecting fraud. By ensuring that either the customer or merchant is always online, fraudulent transactions can be detected and prevented in real time.

11. Atomic Swaps and Refunds

Our solution supports **Camenisch-style atomic swaps**, ensuring that payments and merchandise exchanges are conducted fairly and securely, with the presence of a trusted third party. Additionally, we offer a **refund mechanism** that allows customers to reclaim spent coins under certain conditions, while maintaining anonymity for legitimate users. This is particularly valuable for merchants and consumers seeking transparency and flexibility.

1.2 Roadmap to Read the Book

- **Chapter 2:** This chapter outlines the high-level design of the solution and compares various theoretical solutions with the practical usage of present indian payement systems.
- **Chapter 3:** This chapter describes the implementation of our solution and evaluates its performance and scalability.
- **Chapter 4:** This chapter discusses future work and identifies gaps that need to be addressed for deploying our solution in a production environment.
- **Chapter 5:** The book concludes with an overview of the potential impact and practical relevance of this work.
- **Chapter 6: Bonus** This chapter introduces the cryptographic principles utilized (which can be skipped by experts in cryptography), defines the security properties for income transfer that our system employs, and explains how anonymous e-cash functions. The cryptographic protocols underlying our solution are detailed, and proofs are provided to demonstrate that our protocols satisfy the previously defined security properties.

Chapter 2

High-Level Design

This chapter gives the high level design of the System of our solution.

2.1 Design of the system

Our solution is based on the idea of traditional Chaumian e-cash described in [Cha83], with some differences and additions.Other variants and extensions of anonymous e-cash and blind signatures are discussed in Section 2.3.1.

2.1.1 Entities & External Trusted Support Systems

These are the following Entities of the system :

- **Exchanges:** The exchanges serve as payment service providers for financial transactions between customers and merchants. They hold bank money in escrow in exchange for anonymous digital coins.

- **Customers:** Customers keep e-cash in their electronic wallets and can initiate transactions with merchants by using these digital coins.

- **Merchants:** Merchants accept digital coins in exchange for digital or physical goods and services. The digital coins can be deposited with the exchange in exchange for bank money.

- **Banks:** Banks receive wire transfer instructions from customers and exchanges. It is not necessary for a customer, merchant, and exchange involved in a single payment to have accounts with the same bank, provided that wire transfers can be made between the respective banks.

- **Auditors:** Auditors, typically operated by trusted financial regulators, monitor the behavior of exchanges to assure customers and merchants that the exchanges operate correctly.

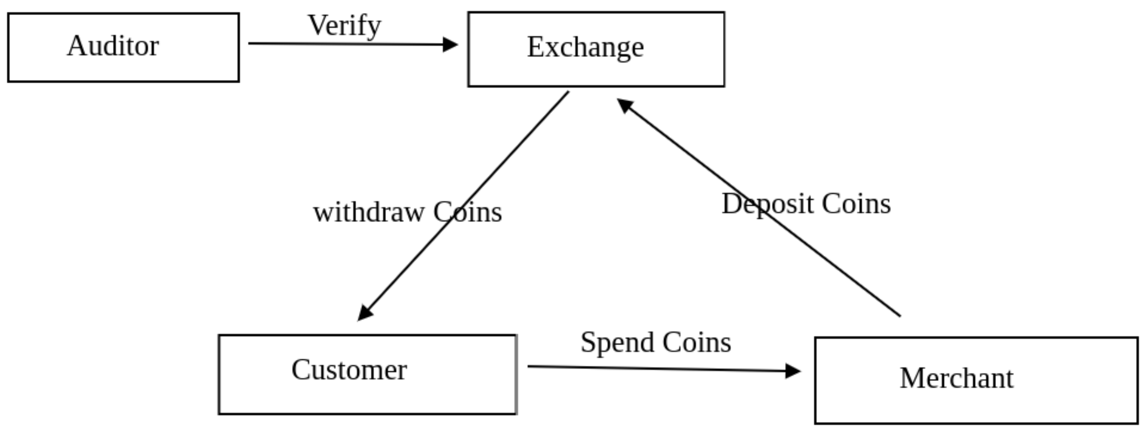


Figure 2.1: HLD of the system without showing the banks and UPI transactions

To pay a merchant, a customer must hold coins at a trusted exchange. To simplify trust, merchants and customers can automatically accept all exchanges audited by a specific auditor. The exchange holds customers' funds in escrow and makes payments to merchants upon digital coin deposits. Customers and merchants gain assurance about the exchange's liquidity and operations through the auditor, typically managed by financial regulators or trusted third parties.

2.1.2 Assumptions of the System

We assume the use of an anonymous, bi-directional communication channel for all interactions between the customer and the merchant. This includes obtaining unlinkable change for partially spent coins from the exchange and retrieving the exchange's public keys necessary for verifying and blindly signing coins. Notably, the withdrawal protocol does not require an anonymous channel to maintain the anonymity of electronic coins.

During the withdrawal process, the exchange identifies the customer, as laws or bank policies restrict the amount of cash an individual can withdraw within a specified time period [Bad15], [Reu15]. Therefore, our solution maintains anonymity only concerning payments. While the exchange knows its customer (KYC), it cannot link the customer's identity to subsequent purchases made at the merchant.

Customers can make untraceable digital cash payments; however, the exchange will always know the merchant's identity, which is necessary for crediting their accounts. This information is also valuable for taxation purposes, as our solution deliberately reveals these transactions as anchors for tax audits on merchants' income. Importantly, while our solution facilitates taxation, it does not implement automatic taxation mechanisms.

We assume that each participant has full control over their system. Both the customer and the merchant are expected to have prior knowledge of the exchange's contact information. Additionally, the customer should be able to authenticate the merchant, for instance, by using X.509 certificates [Yee13]. It is expected that a merchant in our system will deliver the agreed-upon service or goods upon receiving payment. The customer can pursue legal remedies if the merchant fails to fulfill their obligations, as they will have cryptographic evidence of the contract and associated payment.

2.1.3 Reserves

A reserve refers to a customer's non-anonymous funds at an exchange, identified by a unique reserve public key. When a customer wishes to convert money into anonymized digital coins, they first generate a reserve private/public key pair and then transfer funds via their bank to the exchange. The wire transfer instruction must include the reserve public key.

Typically, each wire transfer uses a fresh reserve public key, creating a new reserve. However, using the same reserve public key for another wire transfer merely adds funds to the existing reserve. Even after all funds have been withdrawn, customers should retain the reserve key pair until all coins from the corresponding reserve have been spent. This is crucial because, in the event of a denomination key revocation (see Section 2.2.1), the customer needs this key to recover coins of revoked denominations.

The exchange automatically transfers back to the customer's bank account any funds left in a reserve for an extended period, enabling customers who lost their reserve private key to recover their funds eventually. If a wire transfer to the exchange does not include a valid reserve public key, the exchange will return the money to the sender.

Figure 2.3 illustrates the state machine for a reserve. Long-term states are shown in boxes, while actions are represented by circles. The final state is indicated with a double circle. A reserve is initially filled by a wire transfer, and the balance is decreased by withdrawal operations. If the balance reaches zero, the reserve is considered drained. If a reserve is not drained within a specified timeframe, it is automatically closed. Additionally, a reserve can be refilled through a recoup action (see Section 2.2.1) if the denomination of an unspent coin withdrawn from the reserve is revoked.

Instead of requiring customers to manually generate reserve key pairs and input them into a wire transfer form, banks can offer seamless integration with our solution's wallet software. In this scenario, the bank's website or app provides a "withdraw to wallet"

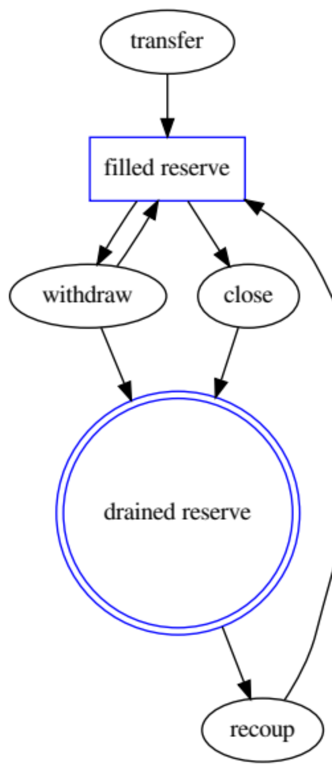


Figure 2.2: State Machine for a Reserve

action. After selecting this action, the user chooses the withdrawal amount from their bank account into the wallet. The bank then instructs the wallet software to create a record of the corresponding reserve, which includes the anticipated amount, the reserve key pair, and the URL of the selected exchange.

When invoked by the bank, the wallet prompts the customer to select an exchange and confirm the reserve creation. The chosen exchange must support the wire transfer method used by the bank, which the wallet checks automatically. Usually, a default exchange is suggested by the bank, and the wallet may also have a predefined list of trusted exchange providers. Subsequently, the wallet returns the reserve public key and the bank account information of the selected exchange to the bank. The bank, typically after verifying a second authentication factor from the customer, will then initiate a wire transfer to the exchange with the information obtained from the wallet.

In cases where the customer's bank does not offer tight integration with our solution, the customer can manually instruct their wallet to create a reserve. The public key must then be included in a bank transaction to the exchange. If the banking app supports pre-filling wire transfer details from a URL or QR code, the wallet can generate such a link or code that includes the pre-filled bank account details of the exchange and the reserve public key. The customer can click on this link or scan the QR code to open their banking app with the pre-filled transaction details. Since there is currently no standardized format for pre-filled wire transfer details, we propose using the 'payto://' URI format, as explained in Section 4.2.1, which is currently under review for acceptance as an IETF Internet Standard.

2.1.4 Coins and Denominations

In our solution, unlike traditional Chaumian e-cash, a coin is represented as a public/private key pair, with the private key known solely to the coin's owner. The financial value of a coin is derived from a blind signature applied to its public key. The exchange offers multiple denomination key pairs for blind-signing coins of varying financial values. Alternative methods for representing different denominations are discussed in Section 2.3.1.

Each denomination key has an expiration date, by which any coins signed with it must either be spent or exchanged for newer coins using the refresh protocol outlined in Section 2.1.6. This mechanism allows the exchange to eventually discard records of old transactions, thereby minimizing the amount of data that needs to be retained and searched to detect double-spending attempts. In the event that a denomination's private key is compromised, the exchange can detect this by monitoring the redemption of coins; if more coins are redeemed than were originally signed into existence using that denomination key, a compromise is indicated. In such cases, the exchange allows verified customers to redeem their unspent coins signed with the compromised private key while rejecting further deposits involving coins signed by that key (see Section 2.2.1). Consequently, the financial impact of losing a private signing key is limited to the amount originally signed with that key, and periodic rotation of denomination keys helps to mitigate this risk.

To prevent the exchange from deanonymizing users by signing each coin with a new denomination key, exchanges publicly announce their denomination keys in advance, along with validity periods that ensure sufficiently strong anonymity sets. These announcements are expected to be signed with the exchange's long-term private master signing key and the auditor's key. Customers should acquire these announcements through an anonymous communication channel.

After a coin is issued, the customer is the sole entity aware of the coin's private key, establishing their ownership. Due to the use of blind signatures, the exchange does not learn the public key during the withdrawal process. If the private key is shared with others, those individuals become co-owners of the coin. The knowledge of the coin's private key and the signature over its public key by the exchange's denomination key are necessary for spending the coin.

2.1.5 Partial Spending and Unlinkable Change

In our solution, customers are not required to have exact change available when making a payment. In fact, it is encouraged to withdraw a larger amount of e-cash in advance, as this blurs the correlation between the non-anonymous withdrawal event and the subsequent anonymous spending event, thereby increasing the anonymity set.

When a customer spends a coin at a merchant, they cryptographically sign a deposit permission using the coin's private key. This deposit permission includes the hash of the contract terms, detailing the purchase agreement between the customer and the merchant. Coins can be partially spent, and the deposit permission specifies the fraction of the coin's value to be paid to the merchant. As digital coins can be easily copied, the merchant must immediately deposit them with the exchange to receive a deposit confirmation or an error indicating a double-spending attempt.

Once a coin is used in a completed or attempted/aborted payment, its public key is revealed to the merchant or exchange. Further payments using the remaining balance would then be linkable to the initial spending event. To enable unlinkable change for a partially spent or otherwise revealed coin, our solution introduces a refresh protocol, consisting of three steps: melt, reveal, and link. This refresh protocol allows the customer to obtain new coins for the remaining amount on a coin. After melting the old coin, it is marked as spent, and the reveal step generates the new coins. By using blind signatures during the withdrawal of the refreshed coins, we ensure they are unlinkable from the original coin.

2.1.6 Refreshing and Taxability

One of the goals of our solution is to ensure that merchants' income remains transparent to state auditors, facilitating appropriate taxation. However, if implemented naively, a simple refresh protocol could be exploited to evade taxes. In this scenario, the recipient of an untaxed transaction could generate the private keys for the coins resulting from refreshing a partially spent old coin and send the corresponding public keys to the payer. The payer would then execute the refresh protocol, provide the payee's coin public keys for blind signing, and send the signatures back to the payee, who would gain exclusive control over the new coins.

To mitigate this risk, our refresh protocol introduces a linking mechanism: coins are refreshed in a manner that allows the owner of the old coin to always obtain the private key and the exchange's signature on the new coins resulting from the refreshes, utilizing a separate linking protocol. This creates a deterrent for merchants attempting to conceal untaxed income. Until the coins are finally deposited at the exchange, the customer retains the ability to regain ownership of them and can deposit them before the merchant has an opportunity to do so. This discourages the circulation of unreported income among untrusted parties within the system.

In our implementation of the refresh and linking protocols, there exists a non-negligible chance of success ($\frac{1}{\kappa}$, where κ is a system parameter, typically ≥ 3) for attempts to cheat during the refresh protocol, resulting in refreshed coins that cannot be recovered from the old coin via the linking protocol. However, cheating during the refresh process is still not profitable, as an unsuccessful attempt leads to a total loss of the amount intended for refreshing.

For the purposes of anti-money laundering and taxation, a more detailed audit of the merchant's transactions may be necessary. A government tax authority can request the merchant to disclose the business agreement details that correspond to the contract terms hash recorded with the exchange. If a merchant fails to provide these values, they may face financial penalties from the state based on the amount transferred through traditional currency methods.

2.1.7 Transactions vs. Sharing

In our solution, sharing differs from a transaction in that it occurs when mutually trusted parties simultaneously possess access to the private keys and signatures of coins. Unlike a transaction, sharing does not establish a clear transfer of control over the funds; instead, both parties maintain equal authority over the shared assets. A practical application of sharing within our framework is peer-to-peer payments between trusted parties, such as family members or friends. This feature enhances the flexibility of transactions while preserving the necessary control and trust between participants.

2.1.8 Aggregation

In our solution, merchants can set a deadline for when the exchange must issue a wire transfer to their bank account for each payment. Before this deadline, multiple payments from deposited coins to the same merchant can be aggregated into a single larger payment. This aggregation reduces transaction costs imposed by underlying banking systems, which often charge a fixed fee for each transaction. To further incentivize merchants to select a longer wire transfer deadline, the exchange may implement a fee for each aggregated wire transfer.

Figure 2.3 illustrates the state machine for processing deposits. Long-term states are represented in boxes, while actions are depicted in circles. The final state is indicated with a double-circle. The deposit process begins when a wallet makes a payment, which

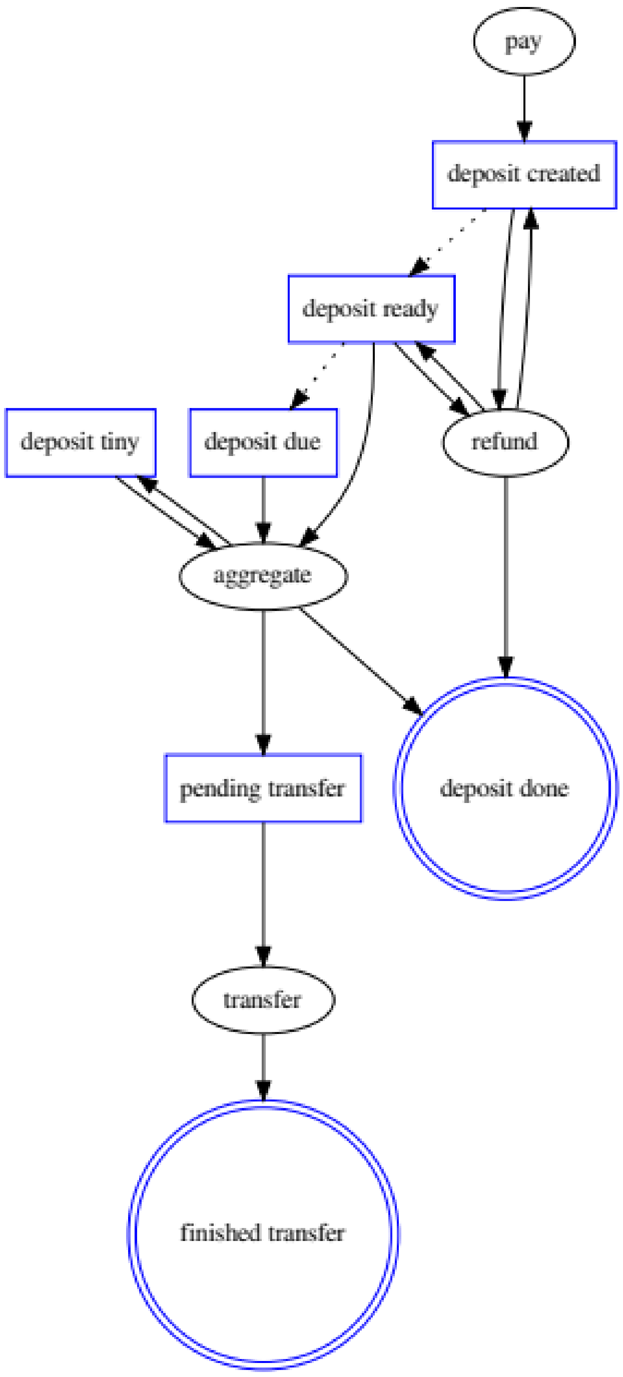


Figure 2.3: State Machine for a deposit

creates a deposit with a refund deadline. The wire transfer cannot occur before this refund deadline. Once the refund deadline has passed, the deposit becomes ready, but it is not automatically wired. Refunds may still be processed at this stage, either fully (resulting in the deposit being canceled) or partially, leaving the remaining value in the same deposit state.

A deposit has a second deadline known as the wire deadline. Once this deadline is reached, the deposit is due and must be aggregated. Aggregation combines all deposits that are due, ready, or tiny into one wire transfer. However, if the aggregated amount remains below the banking system's execution threshold, the deposit enters a special state labeled "tiny" until the aggregated amount meets the required threshold. Once aggregation is complete, the deposits are finalized, and the wire transfer process is initiated, moving through preparation and pending states until the bank confirms the transfer.

2.1.9 Fees

To ensure the sustainable operation of the exchange, our solution incorporates a fee structure for various transactions. The exchange charges fees for withdrawals, refreshing, deposits, and refunds, with the fee amount depending on the denomination used. Different denominations may require varying key sizes, security measures, and storage capacities, leading to differentiated fees.

In our implementation, merchants can choose to absorb fees on behalf of customers, thereby obscuring these costs. Given that various exchanges and denominations may impose different fee structures, merchants can specify a maximum fee amount they are willing to cover. Any fees exceeding this threshold must be directly paid by the customer.

Additionally, the fee structure plays a crucial role in mitigating denial-of-service attacks. To deter adversaries from performing "useless" operations, such as excessive refreshing of coins—which incurs higher storage costs for the exchange—these operations are subject to a fee. Merchants can choose to cover these costs up to a predetermined limit, effectively shielding customers from these fees.

Wire transfer fees are another important aspect of our solution. The exchange charges a fee for each wire transfer to a merchant to offset the costs incurred in the underlying banking system. By structuring these fees, merchants are encouraged to select longer aggregation periods, as the fee is applied per transaction, regardless of the amount transferred.

Moreover, merchants can define the maximum wire transfer fee they are willing to cover for customers, along with an amortization rate for these fees. If the wire fees associated with a payment exceed the specified maximum, the customer must pay the excess fee, distributed according to the amortization rate. Merchants should set this rate based on their expected number of transactions during each wire transfer aggregation window, allowing for better financial management of anticipated wire fees.

2.1.10 The Withdraw Loophole and Tipping

In our solution, the withdrawal protocol can potentially be exploited to transfer funds illicitly. This occurs when a receiver generates a coin's secret key and provides the corresponding public key to the party executing the withdrawal protocol. We refer to this as the "withdraw loophole." However, this exploit is limited to a single transaction, as the properties of the refresh protocol prevent money from circulating among mutually distrusted parties.

A legitimate use of the withdraw loophole is tipping, where merchants reward customers with small amounts (for instance, for completing surveys or installing applications) without any contractual obligations or digitally signed agreements.

Fixing the Withdraw Loophole

To mitigate the misuse of the withdraw loophole for untaxed transactions, we propose the following approach: Standard withdrawal operations and unregistered reserves should be disabled, except for specific tip reserves that merchants register as part of a tipping campaign. Customers would need to pre-register with the exchange and obtain a special withdrawal key pair by providing a small safety deposit. New coins can then be acquired via a refresh operation from the withdrawal key to a new coin.

If customers attempt to exploit our system for untaxed payments, they face the risk of losing money by providing false information during the refresh protocol or by sharing their reserve private key with the payee. To discourage the latter, the exchange would award the safety deposit to the first participant who reports the corresponding private key being used in an illicit transaction, necessitating a new safety deposit before the customer can withdraw again.

However, since the withdraw loophole allows only one additional payment (without any cryptographic evidence for dispute resolution) before the coin must be deposited, the need for these additional mitigations may be questionable, especially considering the costs associated with implementing them.

2.2 Auditing

In our solution, the auditor is a crucial component designed to be deployed by a financial regulator. It serves several key functions to ensure the integrity and compliance of the exchange with relevant regulations:

- It regularly examines the exchange's database and bank transaction history to identify any discrepancies that could indicate fraud or mismanagement.

- It accepts samples of specific protocol responses received by merchants from the audited exchange, verifying that the signatures issued by the exchange correspond to the data stored in its database.
- It certifies exchanges that meet the operational and financial standards required by regulators, ensuring that only compliant exchanges operate within the system.
- It conducts regular anonymous checks to confirm that the necessary protocol endpoints of the exchange are accessible and functioning properly.
- In certain deployment scenarios, merchants must pre-register with exchanges to comply with know-your-customer (KYC) requirements. The auditor provides a list of certified exchanges to merchants, streamlining the KYC registration process.
- It offers customers an interface to submit cryptographic proof if an exchange misbehaves. For instance, if a customer claims that the exchange has denied service, the auditor can execute a request on behalf of the customer to address the issue.

2.2.1 Exchange Compromise Modes

In our solution, the exchange is a critical component that can be an attractive target for hackers and insider threats. We discuss the various ways the exchange can be compromised, methods to reduce the likelihood of such compromises, and strategies for detection and response if they occur.

Compromise of Denomination Keys and Revocation

If a denomination key pair is compromised, an attacker can illegitimately create new coins by using it to sign coins of that denomination. Our solution incorporates mechanisms for the exchange (or its auditor) to detect such compromises. For instance, if the number of deposits for a specific denomination exceeds the number of withdrawals for that same denomination, it raises a red flag.

To mitigate risks, our system allows the exchange to revoke denomination keys, with wallets periodically checking for such revocations. A coin associated with a revoked denomination is referred to as a revoked coin. Once a denomination key is revoked, new coins of that denomination cannot be withdrawn or used for refresh operations. Revoked coins cannot be spent and can only be refreshed if their public keys were recorded in the exchange's database prior to revocation.

The recoup protocol specifies the following cases for handling revoked coins:

1. **Unseen Revoked Coin**: If a revoked coin has never been seen by the exchange but the customer can prove, through a withdrawal protocol transcript and blinding factor, that it originated from a legitimate withdrawal, the exchange credits the respective reserve with the value of the revoked coin.

2. **Partially Spent Coin**: If the coin has been partially spent, the exchange allows the remaining amount to be refreshed into new coins of non-revoked denominations.

3. **Coin from Refresh Protocol**: If the revoked coin C_R has never been seen by the exchange, was obtained via the refresh protocol, and the exchange has a record of either a deposit or refresh for its ancestor coin C_A, the customer can prove this by showing a corresponding refresh protocol transcript and blinding factors. In this case, the exchange credits the remaining value of C_R to C_A. It is permitted for C_A to also be revoked, allowing the customer to obtain their funds by refreshing C_A.

These rules limit the maximum financial damage incurred by the exchange from a compromised denomination key D to $2nv$, where n is the maximum number of D-coins in circulation and v is the financial value of a single D-coin. If denomination D was withdrawn n times by legitimate users, the exchange must immediately revoke D once it detects more than n pairwise different D-coins. An attacker could then gain at most nv by refreshing into non-revoked denominations or spending forged D-coins. Legitimate users can subsequently request a recoup for their coins, resulting in total financial damage limited to $2nv$.

With one rare exception, the recoup protocol does not negatively impact customer anonymity. Specifically, in case (1), the coin obtained from the credited reserve is blindly signed; in case (2), the refresh protocol ensures unlinkability of the non-revoked change; and in case (3), C_R is treated as fresh. Anonymity could be compromised if C_R from case (3) has been seen by a merchant in an aborted transaction; however, such cases should be rare in practice.

Unlike most operations, the recoup protocol incurs no transaction fees. Its primary purpose is to limit financial loss in cases where audits reveal that the exchange's private keys were compromised and to automatically repay balances held in customer wallets if the exchange ceases operations.

To minimize the impact of a compromise, the exchange can utilize a hardware security module (HSM) to store denomination secret keys, which can be pre-programmed with a limit on the number of signatures it can produce. This limit may be mandated by certain auditors who will also audit the operational security of the exchange during the certification process.

Compromise of Signing Keys

In our solution, when a signing key is compromised, an attacker can impersonate a merchant and forge deposit confirmations. To successfully forge a deposit confirmation, the attacker must also obtain a customer's valid signature on a contract that includes the adversary's banking details. The key vulnerability lies in the fact that the customer may have already spent the coin. Consequently, any deposit involving the forged confirmation would be rejected by the exchange due to double spending.

This scenario allows the attacker to claim in court that they properly deposited the coin first and demand payment from the exchange. Furthermore, a malicious exchange could simply choose not to record deposit permissions in its database, subsequently failing to execute them. Thus, when a merchant presents a deposit confirmation, we need a method to determine whether the situation involves a malicious exchange that should be compelled to pay, or a compromised signing key where payouts—and thus financial damage to the exchange—can be justifiably limited.

To mitigate the financial damage from a compromised signing key, merchants are required to collaborate with auditors to conduct probabilistic deposit auditing of the exchange. The aim is to detect signing key compromises by ensuring that the exchange accurately records deposit confirmations. However, verifying each deposit confirmation against the exchange's database would be prohibitively expensive and time-consuming. Fortunately, a probabilistic approach suffices, where merchants only send a small fraction of their deposit confirmations to the auditor. If the auditor encounters a deposit confirmation not recorded in the exchange's database (potentially after synchronizing with the exchange), it indicates that the signing key may have been compromised.

At this point, the signing key must be revoked, and the exchange is obligated to investigate the security of its systems and rectify the issue before resuming normal operations. Nonetheless, various parties (including the attacker) could still present deposit confirmations signed by the revoked key, claiming that the exchange owes them for their deposits. Simply revoking a signing key does not absolve the exchange of its payment obligations, as the attacker may have generated an unlimited number of deposit confirmations with the compromised key.

In contrast to honest merchants, the attacker would not have participated proportionately in the auditor's probabilistic deposit auditing scheme for those deposit confirmations; had they done so, the key compromise would likely have been detected and the key revoked. The exchange is still required to pay all deposit permissions it signed for coins that were not double spent. However, for coins where multiple merchants claim possession of a deposit confirmation, the exchange will compensate the merchants in proportion to the fraction of coins they reported to the auditor as part of the probabilistic deposit auditing.

For instance, if the protocol requires reporting 1% of deposits to the auditor, a merchant may receive payment for at most $100 + X$ times the number of reported deposits, where $X > 0$ ensures proper payouts despite the probabilistic nature of the reporting. This incentivizes honest merchants to accurately report deposit confirmations to the auditor.

With this scheme, the attacker can only report a limited number of deposit confirmations to the auditor before triggering the detection of the signing key compromise. Assuming again that 1% of deposit confirmations are reported by honest merchants, the attacker can only expect to submit around 100 deposit confirmations generated by the compromised signing key before being detected. The anticipated financial gain for the attacker from the key compromise would be $(100 + X) \cdot 100$ deposit confirmations.

Thus, the financial advantage for the attacker can be constrained by probabilistic deposit auditing, while honest merchants maintain strong incentives to engage in the process.

Compromise of the Database

In our solution, if an adversary were to modify the exchange's database, such actions would likely be detected swiftly by the auditor, provided the database employs robust integrity mechanisms. An attacker might also attempt to prevent database updates, thereby blocking the recording of spend operations and executing double spends. This scenario is effectively equivalent to the compromise of signing keys, and can be detected using similar strategies.

Compromise of the Master Key

If the master key is compromised, an attacker could potentially de-anonymize customers by associating different sets of denomination keys with each individual customer. In a properly audited exchange, this malicious activity would be detected promptly, as the denomination keys would not receive validation from auditors.

2.2.2 Cryptographic Proof

In our solution, we utilize the term "proof" to describe the cryptographic assurances provided by the protocol regarding the correct or incorrect behavior of various parties involved. However, as highlighted by [MA14], financial systems must offer evidence that is admissible in legal contexts. Our implementation is designed to generate and export such evidence while adhering to the fundamental principles outlined in [MA14]. Notably, by providing cryptographic proofs as evidence, participants are not required to disclose their core secrets, thus maintaining privacy and security throughout the process.

2.2.3 Perfect Crime Scenarios

In our solution, we have slightly modified the our solution framework to mitigate risks associated with blackmailing or kidnapping attempts by criminals seeking to exploit the anonymity properties of the system to demand ransom payments in anonymous e-cash. These modifications introduce a minor latency penalty for customers during normal usage and require additional data storage within the exchange's database. Therefore, these measures are best suited for deployments where enhanced resistance against perfect crime scenarios is critical, while simpler systems, such as a school cafeteria payment solution, may not require such extensive safeguards.

The following modifications have been implemented:

1. Coins can now be exclusively used in either transactions or refresh operations, but not both simultaneously. This change necessitates that the customer's wallet prepares exact change through the refresh protocol before any transaction occurs. Consequently, transactions will be conducted using precise amounts. This adjustment is vital for maintaining anonymity in light of the second modification, though it does lead to increased storage requirements and latency.

2. The recoup protocol has been revised so that coins obtained through refreshing must be recovered differently if revoked. To recover a revoked coin acquired through refreshing, the customer must present the transcripts for the entire chain of refresh operations, along with the initial withdrawal operation (including the blinding factor). Refreshing on revoked coins is no longer permitted.

Following the payment of ransom to an attacker, the exchange will revoke all currently offered denominations and register a new set of denominations with the auditor. Reserves used to satisfy the ransom will be marked as blocked in the exchange's database. Normal users can utilize the recoup protocol to reclaim funds previously held in revoked denominations. Although the attacker may attempt to recover funds via the modified recoup protocol, this effort will fail due to the blocking of the initial reserve. While the criminal could try to anonymously spend the e-cash before it is revoked, this is likely challenging for substantial amounts. Additionally, due to income transparency, all transactions occurring between the ransom payment and the revocation can be traced back to merchants who may be complicit in laundering the ransom payment.

Honest customers will always be able to use the recoup protocol to transfer their funds back to the original reserve. Because of the first modification, the unlinkability of transactions remains intact, as only coins that were purely utilized for refreshing can now be correlated.

We believe that our approach is more practical than those relying on tracing, as an attacker could always request a plain blind signature in a tracing scheme. Conversely, our approach ensures that the attacker will always lose funds that cannot be immediately spent. However, it is important to note that our method is limited to kidnapping scenarios and is not applicable in blackmail situations where the attacker can inflict damage once they discover that their funds have been erased.

2.2.4 Summary

Figure 2.4 illustrates the overall state machine for processing coins within our solution. Long-term states are represented in boxes, while actions are depicted in circles. The final state is indicated by a double-circle. Dashed arrows represent transitions based on timing rather than external actions. The red arrow signifies an action that the exchange permits but should never be executed by wallets, as it would compromise unlinkability.

A coin begins as an unsigned planchet, which can be signed during either the withdrawal protocol or the refresh protocol. The most common scenario involves depositing the freshly minted coin. This payment creates a deposit (see Figure 2.3) and results in either a dirty coin (if the payment is only for a fraction of the coin's value) or a spent coin. A spent coin can be refunded by the merchant, resulting in the creation of a dirty coin. Once the exchange has aggregated a coin and transferred the amount to the merchant, it can no longer be refunded.

A fresh coin may also face key revocation, which leads the wallet to hold a revoked coin. At this stage, the wallet can utilize the recoup protocol to recover the value of the coin. If the coin was derived from a withdrawal operation, the value is added back into the reserve, which is replenished in the process (see Figure 2.3). Conversely, if the coin originated from the refresh operation, the old coin is converted into a zombie coin, which can be refreshed again.

Both dirty coins and fresh coins can be melted. Dirty coins should be automatically melted by the wallet as soon as possible, as this is the best method to utilize them while maintaining unlinkability. Additionally, the wallet should automatically melt any fresh coins that are at risk of their denomination key nearing its (deposit) expiration time. Failure to do so may lead to the expiration of the coins, resulting in a loss for the coin's owner. Dirty coins can also expire if the melt fee surpasses the residual value of the dirty coin.

To melt a coin, the wallet must commit to one or more planchets and demonstrate honesty when the commitment made for the refresh session is checked during the reveal step. If the wallet was honest, the reveal will yield fresh coins.

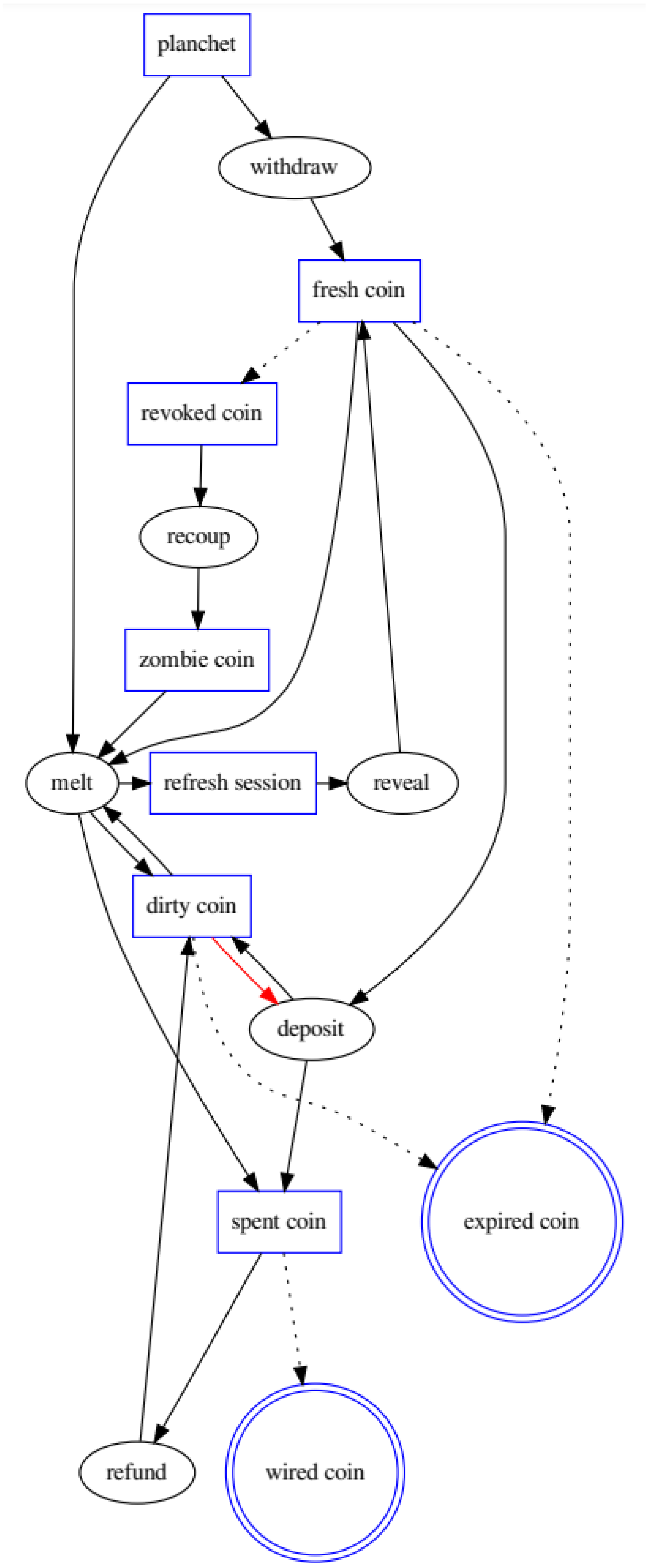


Figure 2.4: State machine of a coin
NPCI x Antaragni Hackathon

Chapter 3

Implementation

This chapter describes the implementation of our solution in detail. Concrete design decisions, protocol details, and our reference implementation are discussed.

We implemented our protocol in the context of a decentralized and secure transaction system, as shown in Figure 2.1. The system was designed for real-world usage with current decentralized technologies and compatibility with financial and secure data systems.

The following technical goals and constraints influenced the design of the protocol and implementation:

- The implementation should allow transactions in environments with hardened security settings. In particular, it must be possible to execute secure transactions without needing complex client-side dependencies such as browser cookies, third-party JavaScript, or external logins.

- Cryptographic evidence should be available to all parties in case of a dispute. All transactions must be verifiable and auditable.

- The implementation must not introduce additional security or privacy vulnerabilities. Features that could compromise privacy for convenience must be clearly communicated, and users must have the option to disable them. Integration with decentralized platforms should minimize the potential for user tracking and data exposure.

- The integration for system participants must be simple. In particular, users should not need to write cryptographic code or manage specific keys and secrets directly within their application. Tools such as Software Development Kits (SDKs) and well-documented APIs must be provided.

- The system should be independent of any specific platform, allowing for a decentralized network with no reliance on a single centralized service. Enhanced features supported by certain platforms are welcome, but graceful fallbacks must exist for other platforms.

- Transaction identifiers and URLs should be clean, user-friendly, and intuitive. This is particularly important when sharing or revisiting them after a session has expired.
- The solution must support multiple currencies and digital assets. Conversion between assets, where applicable, should be handled automatically by the platform, though out of scope for the basic protocol implementation.
- Flexibility is key to supporting different contexts or applications. The implementation must provide hooks for integration with various blockchain and off-chain systems, allowing customization for specific regulatory or operational requirements.
- The implementation must be robust against network failures and crash faults, and recover gracefully from such issues. All operations must be idempotent where possible, ensuring accidental repeated actions do not result in duplicated or failed transactions.
- Authorization should be preferred to authentication wherever possible. Users should not be required to enter sensitive information unless they are interacting with a verifiably secure interface.
- The user experience should be seamless, with no unnecessary flickering or redirects. The number of requests during user navigation, especially during transactions, must be minimized to ensure efficiency.
- The system must support machine-to-machine (M2M) payments and transactions, allowing completely independent operation outside of human interaction. This makes the system applicable for automation, IoT, and decentralized applications (dApps).

3.1 Overview

We provide a high-level overview over the implementation, before discussing the respective components in detail.

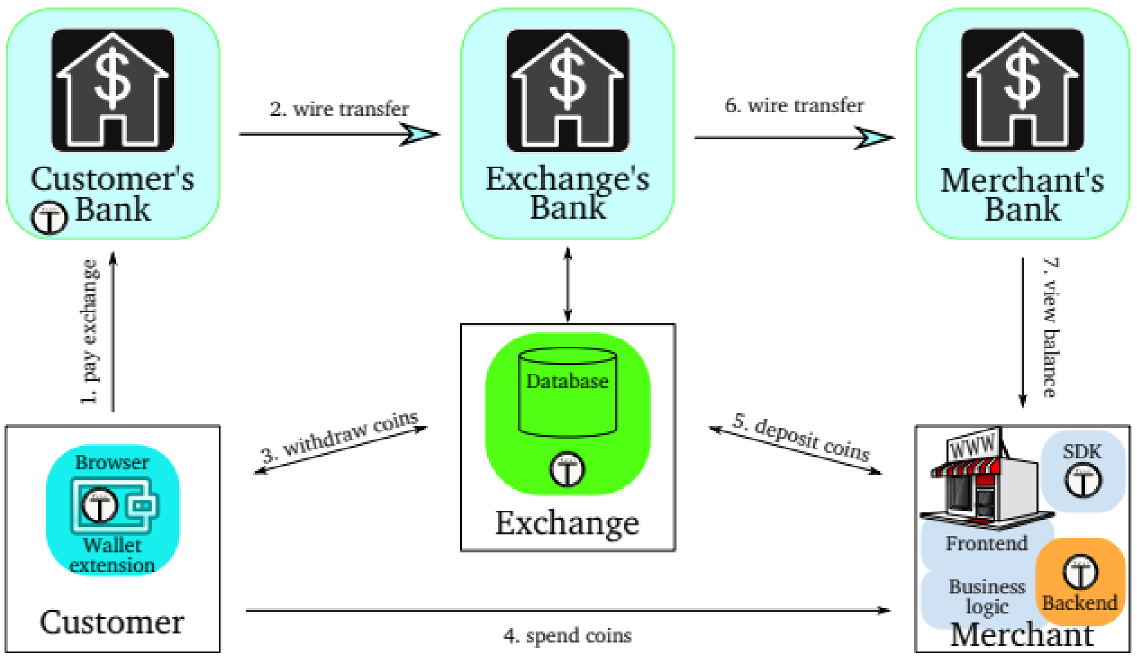


Figure 3.1: The different components of the Our Solution system in the context of a banking system providing money creation, wire transfers and authentication. (Auditor omitted.)

3.1.1 APIs in Our System

The components of our decentralized transaction system communicate over an HTTP-based, RESTful[1] API. All request payloads and responses are JSON[2] documents.

Binary data (such as cryptographic keys, signatures, and hashes) are encoded as base32-crockford[3] strings. Base32-crockford is a simple, case-insensitive encoding of binary data into a subset of the ASCII alphabet that encodes 5 bits per character. While this is not the most space-efficient encoding, it is resilient against transcription errors when manually processed.

Financial amounts are represented as fixed-point decimal numbers. Internally, the system uses a pair of integers (v, f), with value part $0 \leq v \leq 2^{52}$ and fractional part $0 \leq f < 10^8$, to represent an amount $a = v + f \cdot 10^{-8}$. This representation allows for exact handling of financial transactions, with the smallest unit being equivalent to one satoshi (as in Bitcoin), and the largest representable value still fitting within 64-bit IEEE 754 floating-point numbers. This fixed-point representation is essential in financial applications, as it avoids the precision issues inherent in floating-point arithmetic, ensuring

[1]Fielding, Roy Thomas. "Architectural styles and the design of network-based software architectures." (2000)

[2]Bray, Tim. "The JavaScript Object Notation (JSON) Data Interchange Format." (2017)

[3]Crockford, Douglas. "The application/json media type for JavaScript Object Notation (JSON)." (2006)

exact amounts such as 0.10 are handled accurately.

Signatures are created over binary representations of data, with a 64-bit tag that includes the size of the message (32 bits) and an integer tag (32 bits) that uniquely identifies the purpose of the message. When signing JSON objects, a canonical representation is created by removing all whitespace and sorting the object's fields lexicographically. This ensures consistent, verifiable signatures across systems.

In future iterations, more space-efficient formats like BSON[4] or CBOR[5] could be supported. The chosen format can be negotiated between the client and server in a backwards-compatible manner using the HTTP `Accept` header.

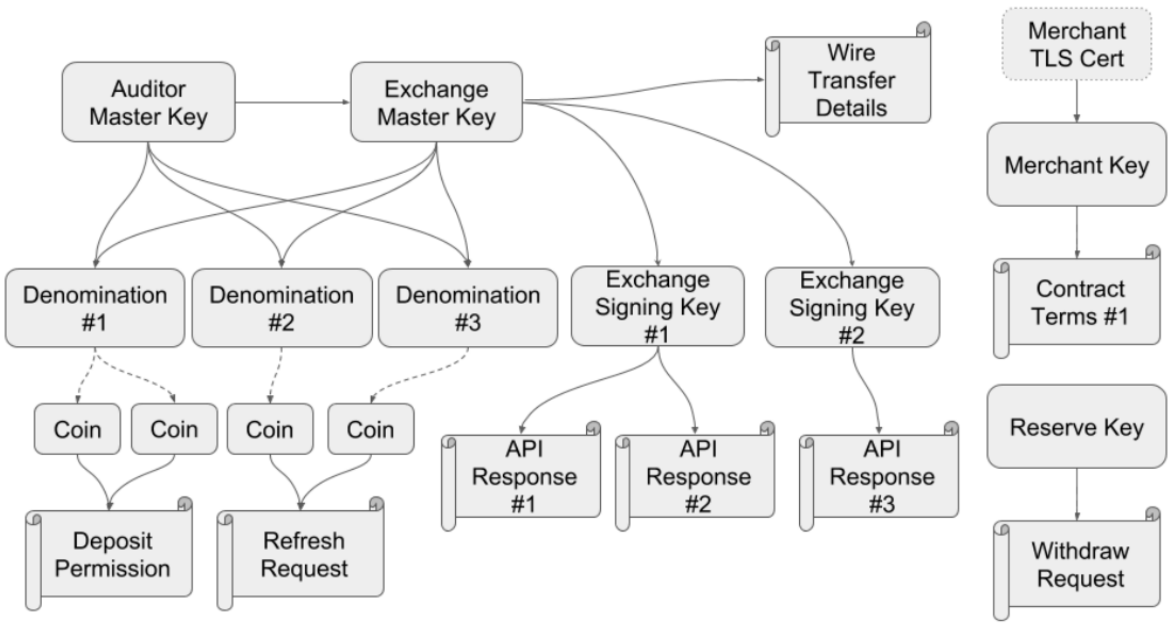


Figure 3.2: Entities/PKI in Our Solution. Solid arrows denote signatures, dotted arrows denote blind signatures.

3.1.2 Cryptographic Algorithms

The following cryptographic primitives are employed in our decentralized transaction system:

- **SHA-512** [H306] as the cryptographic hash function for message integrity and commitment schemes.

[4]MongoDB. "BSON Specification." (2021)
[5]Bormann, Carsten, and Paul Hoffman. "Concise Binary Object Representation (CBOR)." (2013)

- **Ed25519** [Ber06] for signing operations in non-blind signatures, ensuring fast and secure elliptic-curve-based signatures.

- **Curve25519** [Ber06] for Diffie-Hellman key exchange operations and refreshing cryptographic material in transactions.

- **HKDF** [KE10] as the key derivation function, ensuring secure key generation for the refreshing operation and other cryptographic contexts.

- **FDH-RSA blind signatures** [Bel+03] for blind signing operations, preserving user privacy by ensuring the signing process does not reveal transaction details to third parties.

These cryptographic primitives were chosen due to their simplicity, computational efficiency, and rigorous academic scrutiny. Additionally, these algorithms are widely used and trusted in cryptographic applications. Furthermore, alternative cryptographic schemes adhering to the necessary security properties described in Section 3.5.1, such as [**?**], could be employed in place of FDH-RSA.

3.1.3 Entities and Public Key Infrastructure

The public key infrastructure (PKI) employed in our decentralized transaction system is orthogonal to traditional TLS-based PKI . While TLS is utilized as the transport layer for securing API messages, our protocol does not rely on TLS for ensuring the authenticity or integrity of these API queries and responses. However, TLS is critical for ensuring the confidentiality of digital business contracts and safeguarding the authenticity, integrity, and confidentiality of digital product deliveries.

To preserve the anonymity of users, interactions between customers, merchants, and exchanges must occur via an anonymity layer (practical implementations include services like Tor). This ensures that customer identities are protected during transactions, in accordance with the privacy guarantees of our system.

Regarding merchants, we cannot rely on a central trust anchor such as a trusted auditor or the exchange itself for establishing trust. Merchants are not required to register in our PKI system to accept payments. Instead, we leverage TLS: The merchant must include their system-specific public key in their TLS certificate. If a merchant fails to provide this, the system will issue a warning when the user is prompted to confirm a transaction, ensuring the user is informed of any potential authenticity risks.

```
{
  "version": "2:0:0",
  "master_public_key": "CQQZ...",
  "reserve_closing_delay": "/Delay(2419200)/",
  "signkeys": [
    {
      "stamp_start": "/Date(1522223035)/",
      "stamp_expire": "/Date(1533109435)/",
      "stamp_end": "/Date(1585295035)/",
      "master_sig": "842D...",
      "key": "05XW..."
    }
  ],
  "payback": [],
  "denoms": [
    {
      "master_sig": "BHG5...",
      "stamp_start": "/Date(1500450235)/",
      "stamp_expire_withdraw": "/Date(1595058235)/",
      "stamp_expire_deposit": "/Date(1658130235)/",
      "stamp_expire_legal": "/Date(1815810235)/",
      "denom_pub": "51RD...",
      "value": "TESTKUDOS:10",
      "fee_withdraw": "TESTKUDOS:0.01",
      "fee_deposit": "TESTKUDOS:0.01",
      "fee_refresh": "TESTKUDOS:0.01",
      "fee_refund": "TESTKUDOS:0.01"
    },
    {
      "master_sig": "QT0T...",
      "stamp_start": "/Date(1500450235)/",
      "stamp_expire_withdraw": "/Date(1595058235)/",
      "stamp_expire_deposit": "/Date(1658130235)/",
      "stamp_expire_legal": "/Date(1815810235)/",
      "denom_pub": "51R7",
      "value": "TESTKUDOS:0.1",
      "fee_withdraw": "TESTKUDOS:0.01",
      "fee_deposit": "TESTKUDOS:0.01",
      "fee_refresh": "TESTKUDOS:0.01",
      "fee_refund": "TESTKUDOS:0.01"
    },
  ],
  "auditors": [
    {
      "denomination_keys": [
        {
          "denom_pub_h": "RNTQ...",
          "auditor_sig": "6SC2..."
        },
        {
          "denom_pub_h": "CP6B...",
          "auditor_sig": "0GSE..."
        }
      ],
      "auditor_url": "https://auditor.test.taler.net/",
      "auditor_pub": "BW9DC..."
    }
  ],
  "list_issue_date": "/Date(1530196508)/",
  "eddsa_pub": "05XW...",
  "eddsa_sig": "RXCD..."
}
```

Figure 3.3: Example response for /keys

3.1.4 Auditor

Auditors act as trust anchors in our decentralized system. Each auditor is identified by a unique Ed25519 public key. Similar to the certificate store concept in browsers or operating systems, wallet implementations come pre-configured with a list of trusted auditors. These auditors play a crucial role in ensuring system integrity and reliability for participants.

3.1.5 Exchange

An exchange is identified by its long-term Ed25519 master key and the base URL for accessing the exchange. The master key is primarily used as an offline signing key, which is stored on a highly secure, air-gapped machine. All API interactions with the exchange are facilitated by appending the appropriate endpoint to the base URL.

The master key signs the following critical information offline:

- The exchange's online Ed25519 signing keys, which are used to sign responses to API requests. Each key has a limited validity period to ensure security.
- The available denominations, which are fixed amounts that can be used for transactions (discussed further in Section 3.1.6).
- The bank accounts supported by the exchange for withdrawals and deposits, including the associated fees for each account.

To obtain the exchange's signing keys, currently available denominations, and other relevant details, wallets and merchants use the `<base-url>/keys` HTTP endpoint. To optimize traffic, clients can request only the signing keys and denominations that were generated after a specific time. The response from the `/keys` endpoint is signed by the active online signing key, providing customers with verifiable proof in case the exchange attempts to issue inconsistent denomination keys to different users as a means of compromising anonymity.

3.1.6 Coins and Denominations

Denominations are RSA public keys used for signing coins of fixed values using blind signature techniques. The denomination data, signed by the exchange's master key, includes:

- The RSA public key of the denomination.
- The start date, indicating when coins of this denomination can be withdrawn or deposited.
- Fees associated with transactions involving coins of this denomination.
- The **withdraw expiration date**, after which coins can no longer be withdrawn, must be after the start date.
- The **deposit expiration date**, after which coins can no longer be deposited, must be after the withdraw expiration date.
- The **legal expiration date**, after which the exchange can delete all records related to operations with coins of this denomination. This date is typically set well after the deposit expiration date.
- The **fees** associated with the following operations for the coin:
  - Withdraw operation
  - Deposit operation
  - Refresh operation
  - Refund operation

An exchange may be audited by zero, one, or multiple auditors. Each auditor monitors all the denominations currently offered by the exchange; partial audits, where only a subset of denominations is audited, are not supported in the current design.

To enable exchange customers to confirm that the exchange is properly audited, the auditor signs an **auditing request** from the exchange. This request contains basic information about the exchange, as well as all keys offered during the auditing period. In addition to the full auditing request, the auditor also signs an individual **certificate** for each denomination, allowing exchange clients to verify newly offered denominations incrementally.

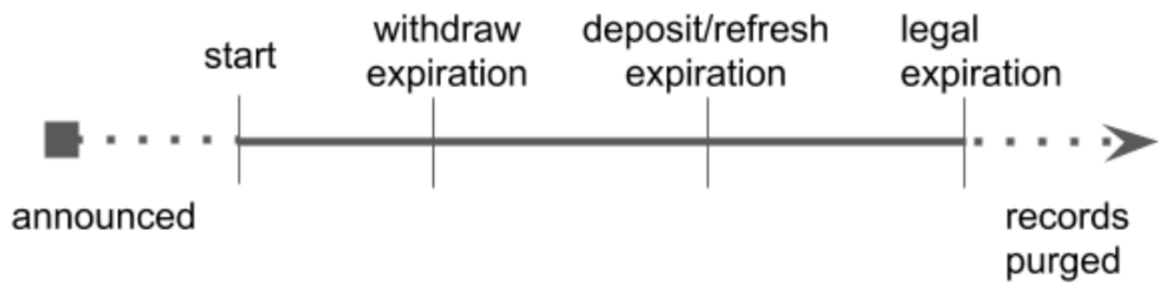


Figure 3.4: A denomination's lifetime.

3.1.7 Merchant

The merchant has one Ed25519 key pair, which is used to sign responses to the customer and authenticate certain requests to the exchange. Depending on the legislation governing a particular Our Solution deployment, merchants may not be required to establish a prior relationship with the exchange. Instead, they may send their bank account information during or after the first deposit from a customer payment.

In some jurisdictions, exchanges are required to comply with know-your-customer (KYC) regulations and verify the identity of merchants [Arn+18] using that exchange for deposits. Typically, a merchant's identity only needs to be verified if they exceed a specific transaction threshold over a given time span. Since the KYC process can be costly for the exchange, this requirement may conflict with merchants' ability to accept payments from all exchanges audited by a trusted auditor. This is because KYC registration must be completed separately at each exchange. Nevertheless, complying with legal requirements is essential for running a legitimate payment system.

A merchant is generally configured with a set of trusted auditors and exchanges, enabling them to accept payments with coins from trusted exchanges and denominations audited by a trusted auditor.

To simplify and secure the deployment of Our Solution, the components responsible for managing the merchant's private key and performing cryptographic operations are isolated into a separate service known as the **merchant backend**. This service has a well-defined RESTful HTTP API, similar to the payment gateways commonly used for credit card transactions. The merchant backend can be deployed either on-premise by the online store or by a third-party provider fully trusted by the merchant.

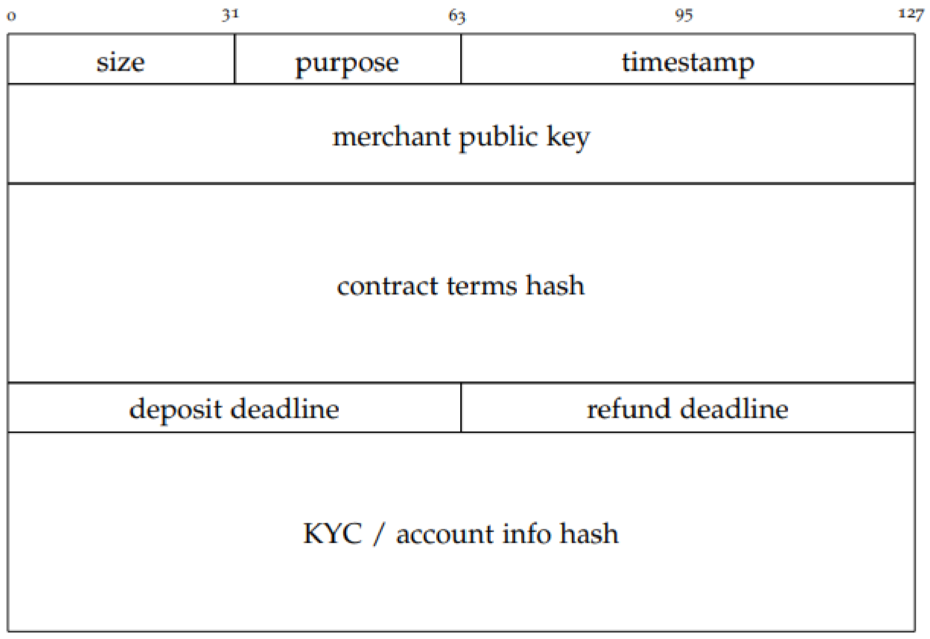


Figure 3.5: The contract header that is signed by the merchant.

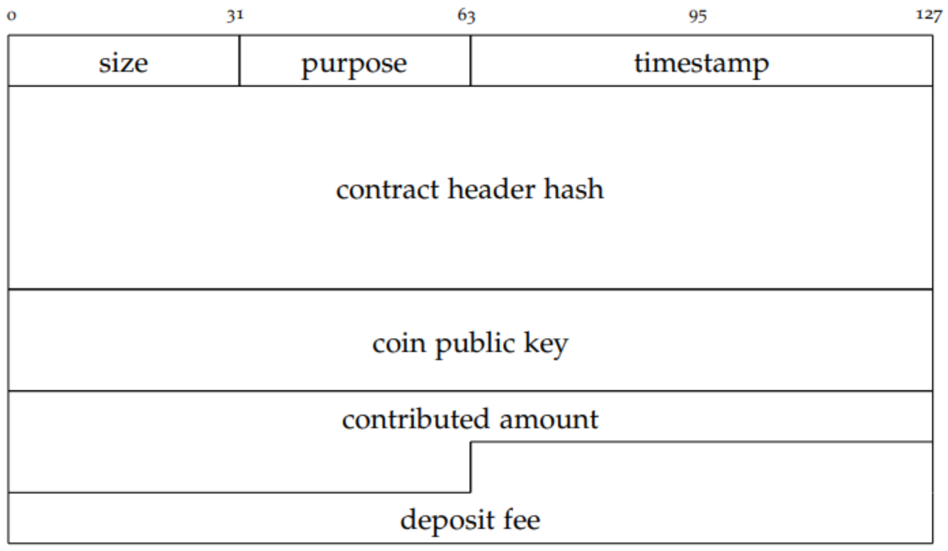


Figure 3.6: The deposit permission signed by the customer's wallet.

3.1.8 Bank

Banks are third parties that are not directly part of our system's Public Key Infrastructure (PKI) and therefore do not participate directly in the PKI used for authentication and communication within the platform. However, they are involved indirectly as external entities for wire transfers and other financial operations.

3.1.9 Customer

Customers are not required to register with an exchange directly. Instead, they use the private keys associated with the reserves they own to authenticate transactions with the exchange. The exchange identifies reserves by their public keys, which are embedded in the subject or instruction data of wire transfers initiated by the customer. Wire transfers that do not include a valid public key are automatically reversed to prevent unauthorized or invalid transactions.

3.1.10 Payments

Payments within our system are based on contract terms, which are represented as a JSON object describing the specifics of the business transaction. The cryptographic hash of this contract object acts as a globally unique identifier for the transaction. Merchants are required to sign the contract terms before sending them to the customer. This allows the customer to prove the merchant's obligations in the event of a dispute.

To preserve privacy, only the merchant and the customer need to know the full content of the contract terms unless a third party becomes involved in a dispute. The exchange, however, must be aware of certain payment-specific details such as refund policies, aggregation deadlines for micropayments, and the merchant's KYC data (typically hashed to prove KYC enrollment).

The merchant's signature covers the contract header, which includes the hash of the contract terms and the payment modalities.

In addition to the merchant-provided data, the contract terms include a `claim_pub` field containing an Ed25519 public key generated by the customer. The customer uses the corresponding private key to prove that they received the contract from the merchant, ensuring they did not duplicate contract terms given to another customer. This key is unique to each transaction and not linked to the customer's permanent identity.

The merchant backend generates the signed contract header based on an order created by the merchant's frontend. The order contains a subset of the contract terms, and the merchant backend automatically includes additional details, such as the merchant's public key, bank account information, and accepted auditors/exchanges. The customer's `claim_pub` is also added by the merchant backend.

An order contains a unique identifier (order ID), which can be human-friendly like a booking number. If not provided manually, the merchant backend automatically generates it. The order ID allows for reference to the payment without knowing the contract terms hash, which is only available after the customer provides their `claim_pub` key.

To initiate a payment, the merchant sends an unclaimed contract URL to the customer. The customer claims the contract by appending their `claim_pub` as a query parameter to this URL and performing an HTTP GET request. The customer verifies that the contract terms are correctly signed by the merchant and contain their `claim_pub`.

To prevent malicious customers from guessing unclaimed contract URLs and claiming contracts, the URLs must contain sufficient entropy, especially for products with limited availability. For example, a ticketing system could send a URL with a nonce to customers on a waiting list, ensuring that an attacker cannot guess the URL and claim tickets before the intended customer.

To finalize the payment, the customer signs a deposit permission for each coin used in the payment. The deposit permission is signed using the coin's private key and includes:

- The amount contributed by the coin,
- The merchant's public key,
- The contract header with the merchant's signature,
- The time at which the deposit permission was signed.

The customer sends the deposit permissions via an HTTP POST request to the `pay_url` specified by the merchant. The merchant deposits each coin with the exchange and, upon successful deposit, provides the customer with a payment confirmation. This confirmation proves that the customer completed the payment within the deadline specified in the contract terms.

Depositing multiple coins does not have transactional semantics, meaning each coin is deposited in a separate transaction. This allows the exchange to scale more efficiently, but requires mechanisms to recover from partially completed payments.

If one of the customer's coins is invalid (e.g., due to a wallet restoration), the customer can retry the payment with a valid coin or request a refund from the merchant. If the merchant refuses a refund or becomes unresponsive, the customer can complete the deposits themselves or use the deposit permissions to prove that the payment was completed.

Allowing customers to request refunds directly from the exchange would require the merchant to include the payment amount in the contract header, but this would expose product prices to the exchange, violating privacy. Therefore, this method is not implemented to maintain confidentiality of business agreements.

3.1.11 Resource-based Web Payments

To seamlessly integrate with the web's architecture, our solution supports payments for web resources, which are represented by URLs. Every contract in our system includes a fulfillment URL that identifies the resource being paid for. If the payment is for a non-digital product (e.g., a physical item or donation), the fulfillment URL may lead to a confirmation page with additional details, such as shipment tracking or donation receipts. The fulfillment URL can either represent a single item or a collection of resources, such as a shopping cart.

Below are the typical steps for processing a payment in our system using the domain `example-shop.com`:

1. The user navigates to a resource on the shop's website, such as `https://example-shop.com/item.`
2. The shop responds with an HTTP status code `402 Payment Required`, including the following headers:
   - `Contract-Url: https://example-shop.com/contract?product=item.pdf`
   - `Resource-Url: https://example-shop.com/item.pdf`
3. If the user's wallet does not contain a contract with the fulfillment URL `https://example-shop.c` the wallet claims the contract by generating a claim key pair (s, p) and requests the contract URL with the claim public key p as a query parameter:

   `https://example-shop.com/contract?product=item.pdf&claim_pub=p`.
4. The wallet displays the contract terms to the user for approval. Once the customer accepts the terms, the wallet initiates a payment to the merchant. The merchant marks the order as paid upon receiving a valid payment.
5. The wallet constructs the extended fulfillment URL by appending the order ID from the contract as an additional parameter, then redirects the browser to:

   `https://example-shop.com/item.pdf?order_id=....`

6. Upon receiving the request to the extended fulfillment URL, the shop verifies the payment's completion status corresponding to the order ID. If the payment is successful, the shop serves the purchased content.

To avoid checking the payment status for every access, the merchant can set a session cookie (signed/encrypted by the merchant) in the user's browser, indicating that the item has already been purchased.

If a customer revisits a resource they previously paid for without the extended fulfillment URL or if the session cookie has expired, the wallet will search for an existing contract that matches the plain fulfillment URL in the merchant's HTTP `402` response. If such a contract is found, the wallet will redirect the user to the corresponding extended fulfillment URL, bypassing the need for a new payment.

In some cases, the fulfillment URL and the payment-triggering URL might differ. For example, if the merchant backend is hosted by a third party like `https://payment-processor.com/`, the payment URL could have a different origin than the fulfillment URL.

This cross-origin behavior poses potential privacy risks. An attacker could attempt to check if a user has paid for a particular resource by sending an HTTP `402` response with the `Resource-Url` set to the resource URL u and the `Contract-Url` pointing to the attacker's server. If the user has already paid for u, the wallet would navigate to the extended fulfillment URL, otherwise, it would attempt to download a new contract from the attacker's server.

To mitigate this attack, the wallet only redirects to the fulfillment URL u if the origin of the `402` response matches the origin of u or the payment URL.

3.1.12 Loose Browser Integration

The payment process outlined previously does not function directly in browsers lacking native integration with our solution. To address this limitation, a fallback mechanism is implemented transparently in our merchant backend.

In addition to signaling that payment is required via the HTTP status code, the merchant includes a fallback URL in the body of the `402 Payment Required` response. This fallback URL utilizes a custom URL scheme and includes the contract terms URL along with other relevant parameters typically conveyed through headers. For instance, the payment process might provide a link (labeled, for example, "Pay with Our Solution") to the following URL, which encodes the same information as the headers:

```
our_solution:pay?
,→ contract_url=
,→ https%3A%2F%2Falice-shop.example.com%2Fcontract%3Fproduct%3D
,→ Dessay-24.pdf
,→ &resource_url=
,→ https%3A%2F%2Falice-shop.example.com%2Fessay-24.pdf
```

This fallback can be disabled for requests from user agents that are known to support our solution natively.

Wallet applications that support our solution register themselves as handlers for the custom URI scheme, enabling users to open the dedicated wallet by following a corresponding link, even when native browser support or extensions are absent. Custom protocol handlers for URI schemes can be registered on all modern platforms with web browsers.

It is important to note that wallets communicate with the merchant from a different browsing context. Consequently, the merchant backend cannot rely on cookies set in the user's browser while using the shop page.

We selected HTTP headers as the primary means of signaling to the wallet instead of using alternative methods (such as a new content media type). This choice allows the fallback content to be rendered as an HTML page compatible with all browsers. Moreover, current browser extension mechanisms permit synchronous interception of headers before rendering begins, thus minimizing any visible flickering associated with intermediate page loads.

3.1.13 Session-bound Payments and Sharing

As we described the payment protocol so far, an extended fulfillment URL is not bound to a browser session. When sharing an extended fulfillment URL, another user would get access to the same content. This might be appropriate for some types of fulfillment pages (such as a donation receipt), but is generally not appropriate when digital content is sold. Even though it is trivial to share digital content unless digital restrictions management (DRM) is employed, the ability to share links might set the bar for sharing too low.

While the validity of a fulfillment URL could be limited to a certain time, browser session or IP address, this would be too restrictive for scenarios where the user wants to purchase permanent access to the content.

As a compromise, Our Solution provides session-bound payments. For session-bound payments, the seller's website assigns the user a random session ID, for example, via a session cookie. The extended fulfillment URL for session-bound payments is constructed by additionally specifying the URL parameter `session_sig`, which contains proof that the user completed (or re-played) the payment under their current session ID.

To initiate a session-bound payment, the HTTP 402 response must additionally contain the `Session-Id` header, which will cause the wallet to additionally obtain a signature on the session ID from the merchant's pay URL, by additionally sending the session ID when executing (or re-playing) the payment. As an optimization, instead of re-playing the full payment, the wallet can also send the session ID together with the payment receipt it obtained from the completed payment with a different session ID.

Before serving paid content to the user, the merchant simply checks if the session signature matches the assigned session and contract terms. To simplify the implementation of the frontend, this signature check can be implemented as a request to the Our backend. Using session signatures instead of storing all completed session-bound payments in the merchant's database saves storage.

While the coins used for the payment or the payment receipt could be shared with other wallets, it is a higher barrier than just sharing a URL. Furthermore, the merchant could restrict the rate at which new sessions can be created for the same contract terms and restrict a session to one IP address, limiting sharing.

For the situation where a user accesses a session-bound paid resource and neither has a corresponding contract in their wallet nor does the merchant provide a contract URL to buy access to the resource, the merchant can specify an offer URL in the `Offer-Url` header. If the wallet is not able to take any other steps to complete the payment, it will redirect the user to the offer URL. As the name suggests, the offer URL can point to a page with alternative offers for the resource, or some other explanation as to why the resource is not available anymore.

3.1.14 Embedded Content

So far we only considered paying for a single, top-level resource, namely the fulfillment URL. In practice, however, websites are composed of many subresources such as embedded images and videos.

We describe two techniques to "paywall" subresources behind a our Solution payment. Many other approaches and variations are possible.

1. Visiting the fulfillment URL can set a session cookie. When a subresource is requested, the server will check that the customer has the correct session cookie set.

2. When serving the fulfillment page, the merchant can add an additional authentication token to the URLs of subresources. When the subresource is requested, the validity of the authentication token is checked. If the merchant itself (instead of a Content Delivery Network that supports token authentication) is serving the paid

subresource, the order ID and session signature can also be used as the authentication token.

It would technically be possible to allow contract terms to refer to multiple resources that are being purchased by including a list or pattern that defines a set of URLs. The browser would then automatically include information to identify the completed payment in the request for the subresource. We deliberately do not implement this approach, as it would require deeper integration in the browser than possible on many platforms. If not restricted carefully, this feature could also be used as an additional method to track the user across the merchant's website.

3.1.15 Contract Terms

The contract terms, only seen by the customer and the merchant (except when a tax audit of the merchant is requested), contain the following information:

- The total amount to be paid,
- the `pay_url`, an HTTP endpoint that receives the payment,
- the deadline until the merchant accepts the payment (repeated in the signed contract header),
- the deadline for refunds (repeated in the signed contract header),
- the claim public key provided by the customer, used to prove they have claimed the contract terms,
- the order ID, which is a short, human-friendly identifier for the contract terms within the merchant,
- the `fulfillment_url`, which identifies the resource that is being paid for,
- a human-readable summary and product list,
- the fees covered by the merchant (if the fees for the payment exceed this value, the customer must explicitly pay the additional fees),
- depending on the underlying payment system, KYC registration information or other payment-related data that needs to be passed on to the exchange (repeated in the signed contract header),
- the list of exchanges and auditors that the merchant accepts for the payment,

- information about the merchant, including the merchant public key and contact information.

3.2 Bank Integration

In order to use Our solution for real-world payments, it must be integrated with the existing banking system. Banks can choose to tightly integrate with Our solution and offer the ability to withdraw coins on their website. Even existing banks can be used to withdraw coins via a manual bank transfer to the exchange, with the only requirement that the 52-character alphanumeric, case-insensitive encoding of the reserve public key can be included in the transaction without modification other than case folding and white space normalization.

3.2.1 Wire Method Identifiers

We introduce a new URI scheme `payto`, which is used in Our Solution to identify target accounts across a wide variety of payment systems with uniform syntax.

In its simplest form, a payto URI identifies one account of a particular payment system:

`payto://` TYPE / ACCOUNT

When opening a payto URI, the default action is to open an application that can handle payments with the given type of payment system, with the target account pre-filled. In its extended form, a payto URL can also specify additional information for a payment in the query parameters of the URI.

In the generic syntax for URIs, the payment system type corresponds to the authority, the account corresponds to the path, and additional parameters for the payment correspond to the query parameters. Conforming to the generic URI syntax makes parsing of payto URIs trivial with existing parsers.

Formally, a payto URI is an encoding of a partially filled out pro forma invoice. The full specification of the payto URI is RFC XXXX.

In the implementation of Our solution, payto URIs are used in various places:

1. The exchange lists the different ways it can accept money as payto URIs. If the exchange uses payment methods that do not have tight integration.

2. In order to withdraw money from an exchange that uses a bank account type that does not typically have tight integration, the wallet can generate a link and a QR code that already contains the reserve public key. When scanning the QR code with

a mobile device that has an appropriate banking app installed, a bank transfer form can be pre-filled and the user only has to confirm the transfer to the exchange.

3. The merchant specifies the account it wishes to be paid on as a payto URI, both in the configuration of the merchant backend as well as in communication with the exchange.

A major advantage of encoding payment targets as URIs is that URI schemes can be registered with an application on most platforms, and will be "clickable" in most applications and open the right application when scanned as a QR code.

3.3 Exchange

The exchange consists of three independent processes:

- The `exchange-httpd` process handles HTTP requests from clients, mainly merchants and wallets.
- The `exchange-wirewatch` process watches for wire transfers to the exchange's bank account and updates reserves based on that.
- The `exchange-aggregator` process aggregates outgoing transactions to merchants.

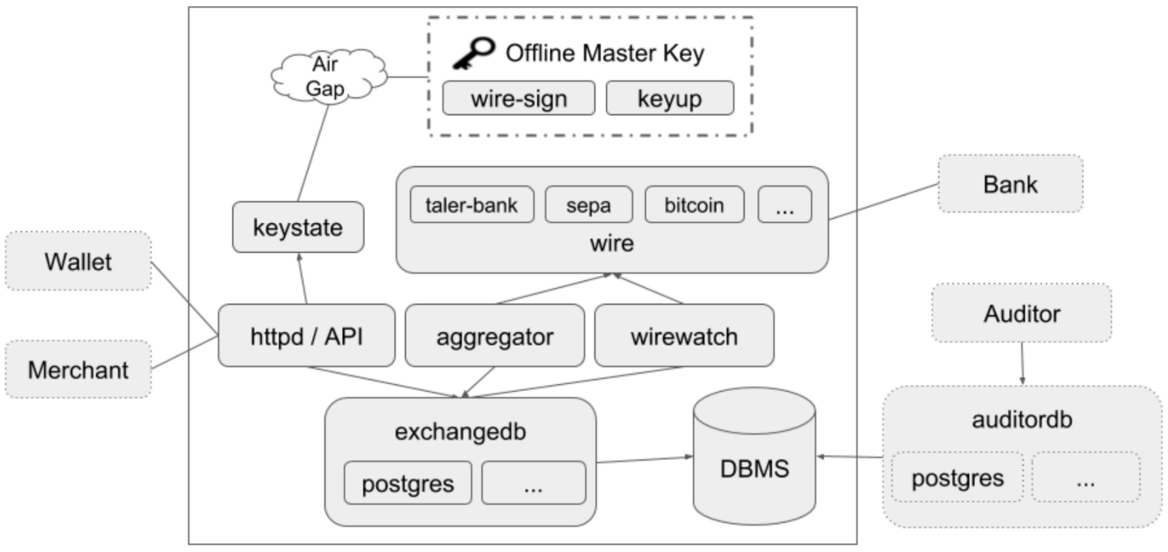


Figure 3.7: Architecture of the exchange reference implementation.

All three processes exchange data via the same database. Only `exchange-httpd` needs access to the exchange's online signing keys and denomination keys.

The database is accessed via a specific database abstraction layer. Different databases can be supported via plugins; at the time of writing this, only a PostgreSQL plugin has been implemented.

Wire plugins are used as an abstraction to access the account layer that our solution runs on. Specifically, the `wirewatch` process uses the plugin to monitor incoming transfers, and the `aggregator` process uses the wire plugin to make wire transfers to merchants.

The following APIs are offered by the exchange:

- **Announcing keys, bank accounts and other public information:** The exchange offers the list of denomination keys, signing keys, auditors, supported bank accounts, revoked keys, and other general information needed to use the exchange's services via the `/keys` and `/wire` APIs.

- **Obtaining entropy:** As we cannot be sure that all client devices have an adequate random number generator, the exchange offers the `/seed` endpoint to download some high-entropy value. Clients should mix this seed with their own, locally-generated entropy into an entropy pool.

- **Reserve status and withdrawal:** After having wired money to the exchange, the status of the reserve can be checked via the `/reserve/$RESERVE_PUB/status` API. Since the wire transfer usually takes some time to arrive at the exchange, wallets should periodically poll this API and initiate a withdrawal with `/reserve/$RESERVE_PUB/withdra` once the exchange received the funds.

- **Deposits and tracking:** Merchants transmit deposit permissions they have received from customers to the exchange via the `/coins/$COIN_PUB/deposit` API. Since multiple deposits are aggregated into one wire transfer, the merchant additionally can use the exchange's `/transfers/$WTID` API that returns the list of deposits for a wire transfer identifier (WTID) included in the wire transfer to the merchant, as well as the `/deposits/$H_WIRE/$MERCHANT_PUB/$H_CAPI` to look up which wire transfer included the payment for a given deposit.

- **Refresh**
  Refreshing consists of two stages. First, using `/coins/$COIN_PUB/melt`, an old, possibly dirty coin is melted and thus devaluated. The commitment made by the wallet during the melt and the resulting γ-challenge from the exchange are associated with a refresh session. Then, using `/refreshes/$RCH/reveal`, the wallet can answer the challenge and obtain fresh coins as change. Finally, `/coins/$COIN_PUB/link` provides the link deterrent against refresh abuse.

- **Refunds**
  The refund API (`/coins/$COIN_PUB/refund`) can “undo” a deposit if the merchant gave their signature, and the aggregation deadline for the payment has not occurred yet.

- **Recoup**
  The recoup API (`/coins/$COIN_PUB/recoup`) allows customers to be compensated for coins whose denomination key has been revoked. Customers must send either a full withdrawal transcript that includes their private blinding factor, or a refresh transcript (of a refresh that had the revoked denominations as one of the targets) that includes blinding factors. In the former case, the reserve is credited; in the latter case, the source coin of the refresh is refunded and can be refreshed again.

New denomination and signing keys are generated and signed with the exchange’s master secret key using the `exchange-keyup` utility, according to a key schedule defined in the exchange’s configuration. This process should be done on an air-gapped offline machine that has access to the exchange’s master signing key.

Generating new keys with `exchange-keyup` also generates an auditing request file, which the exchange should send to its auditors. The auditors then certify these keys with the `auditor-sign` tool.

This process is illustrated in Figure 4.8.

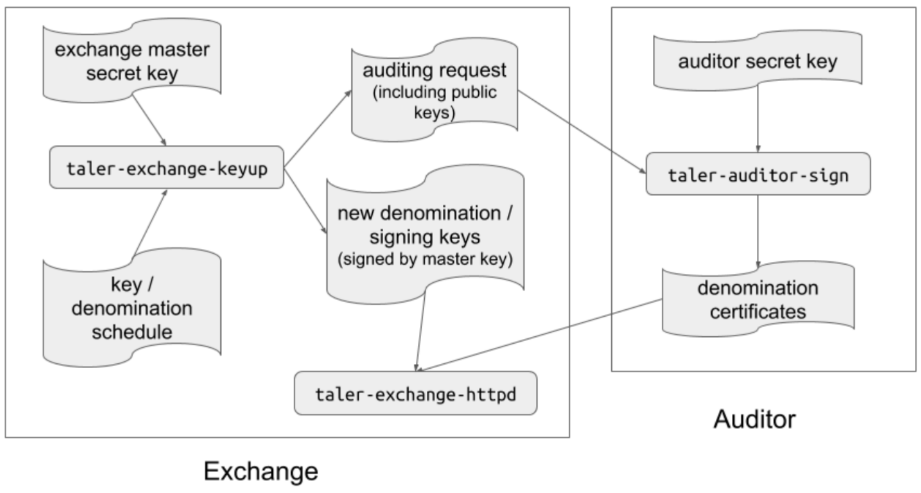

exchange master secret key
auditing request (including public keys)
auditor secret key
taler-exchange-keyup
taler-auditor-sign
new denomination / signing keys (signed by master key)
key / denomination schedule
denomination certificates
taler-exchange-httpd
Auditor
Exchange

Figure 3.8: Data flow for updating the exchange’s keys.

3.4 Auditor

The auditor consists of several main components:

- `auditor-dbinit` tool to setup, upgrade, or garbage-collect an auditor's database,
- `auditor-exchange` tool to add an exchange to the list of audited exchanges,
- `auditor-sign` tool to sign an exchange's keys to affirm that the auditor is auditing this exchange,
- An HTTP service (`auditor-httpd`) which receives deposit confirmations from merchants, and
- The `auditor` script which must be regularly run to generate audit reports.

3.4.1 Database synchronization

FIXME: describe issue of how to synchronize exchange and auditor databases, and how we solved it (once we did solve it!) here.

3.4.2 Auditor tool

The auditor script uses several helper processes. These helper processes access the exchange's database, either directly (for exchange-internal auditing as part of its operational security) or over a replica (in the case of external auditors).

The auditor script ultimately generates a report with the following information:

- Do the operations stored in a reserve's history match the reserve's balance?
- Did the exchange record outgoing transactions to the right merchant for deposits after the deadline for the payment was reached?
- Do operations recorded on coins (deposit, refresh, refund) match the remaining value on the coin?
- Do operations respect the expiration of denominations?
- For a denomination, is the number of pairwise different coin public keys recorded in deposit/refresh operations smaller or equal to the number of blind signatures recorded in withdraw/refresh operations? If this invariant is violated, the corresponding denomination must be revoked.
- What is the income of the exchange from different fees?

Report generation

The auditor script invokes its helper processes, each of which generates a JSON file with the key findings. The master script then uses Jinja2 templating to fill a LaTeX template with the key findings, and runs pdflatex to generate the final PDF.

It is also possible to run the helper processes manually, and given that only one of them requires read-only access to the bank account of the exchange, this may be useful to improve parallelism or enhance privilege separation. Thus, auditor is really only a convenience script.

Incremental processing

The operation of all auditor helper processes is incremental. There is a separate database to checkpoint the auditing progress and to store intermediate results for the incremental computation. Most database tables used by the exchange are append-only: rows are only added but never removed or changed. Tables that are destructively modified by the exchange only store cached computations based on the append-only tables. Each append-only table has a monotonically increasing row ID. Thus, the auditor's checkpoint simply consists of the set of row IDs that were last seen.

Helper Auditor Aggregation

This tool checks that the exchange properly aggregates individual deposits into wire transfers (see Figure 2.3).

The list of invariants checked by this tool thus includes:

- That the fees charged by the exchange are those the exchange provided to the auditor earlier, and that the fee calculations (deposit fee, refund fee, wire fee) are correct. Refunds are relevant because refunded amounts are not included in the aggregate balance.
- The sanity of fees, as fees may not exceed the contribution of a coin (so the deposit fee cannot be larger than the deposited value, and the wire fee cannot exceed the wired amount). Similarly, a coin cannot receive refunds that exceed the deposited value of the coin, and the deposit value must not exceed the coin's denomination value.
- That the start and end dates for the wire fee structure are sane, that is cover the timeframe without overlap or gaps.
- That denomination signatures on the coins are valid and match denomination keys known to the auditor.

- That the target account of the outgoing aggregate wire transfer is well-formed and matches the account specified in the deposit.
- That coins that have been claimed in an aggregation have a supporting history.
- That coins which should be aggregated are listed in an aggregation list, and that the timestamps match the expected dates.

Helper Auditor Coins

This helper focuses on checking the history of individual coins (as described in Figure 2.4), ensuring that the coin is not double-spent (or over-spent) and that refreshes, refunds, and recoups are processed properly.

Additionally, this tool includes checks for denomination key abuse by verifying that the value and number of coins deposited in any denomination do not exceed the value and number of coins issued in that denomination.

Finally, the auditor will also complain if the exchange processes denominations that it did not properly report (with fee structure) to the auditor.

The list of invariants checked by this tool thus includes:

- Testing for an emergency on denominations because the value or number of coins deposited exceeds the value or number of coins issued; if this happens, the exchange should revoke the respective denomination.
- Checking for arithmetic inconsistencies from exchanges not properly calculating balances or fees during the various coin operations (withdraw, deposit, melt, refund);
- That signatures are correct for denomination key revocation, coin denominations, and coin operations (deposit, melt, refund, recoup).
- That denomination keys are known to the auditor.
- That denomination keys were actually revoked if a recoup is granted.
- Whether there exist refresh sessions from coins that have been melted but not (yet) revealed (this can be harmless and no fault of the exchange, but could also be indicative of an exchange failing to process certain requests in a timely fashion).
- That the refund deadline is not after the wire deadline (while harmless, such a deposit makes inconsistent requirements and should have been rejected by the exchange).

Helper Auditor Deposits

This tool verifies that the deposit confirmations reported by merchants directly to the auditor are also included in the database we got from the exchange. This is to ensure that the exchange cannot defraud merchants by simply not reporting deposits to the auditor or an exchange signing key being compromised (as described in Section).

Helper Auditor Reserves

This figure checks the exchange's processing of the balance of an individual reserve, as described in Figure 2.2.

The list of invariants checked by this tool thus includes:

- Correctness of the signatures that legitimized withdraw and recoup operations.
- Correct calculation of the reserve balance given the history of operations (incoming wire transfers, withdraws, recoups, and closing operations) involving the reserve.
- That the exchange closed reserves when required, and that the exchange wired the funds back to the correct (originating) wire account.
- Knowledge of the auditor of the denomination keys involved in withdraw operations and of the applicable closing fee.
- That denomination keys were valid for use in a withdraw operation at the reported time of withdrawal.
- That denomination keys were eligible for recoup at the time of a recoup.

Helper Auditor Wire

This helper process checks that the incoming and outgoing transfers recorded in the exchange's database match wire transfers of the underlying bank account. To access the transaction history (typically recorded by the bank), the wire auditor helper is special in that it must be provided the necessary credentials to access the exchange's bank account. In a production setting, this will typically require the configuration and operation of a Nexus instance (of LibEuFin) at the auditor.

The actual logic of the wire auditor is pretty boring: it goes over all bank transactions that are in the exchange's database, and verifies that they are present in the records from the bank, and then it goes over all bank transactions reported by the bank, and again checks that they are also in the exchange's database. This applies for both incoming and outgoing wire transfers. The tool reports any inconsistencies, be they in terms of wire transfer subject, bank accounts involved, amount that was transferred, or timestamp.

For incoming wire transfers, this check protects against the following failures: An exchange reporting the wrong amount may wrongfully allow or refuse the withdrawal of coins from a reserve. The wrong wire transfer subject might allow the wrong wallet to withdraw, and reject the rightful owner. The wrong bank account could result in the wrong recipient receiving funds if the reserve is closed. Timestamp differences are usually pretty harmless, and small differences may even occur due to rounding or clock synchronization issues. However, they are still reported as they may be indicative of other problems.

For outgoing wire transfers, the implications arising from an exchange making the wrong wire transfers should be obvious.

The list of invariants checked by this tool thus includes:

- The exchange correctly listing all incoming wire transfers.
- The bank/Nexus having correctly suppressed incoming wire transfers with non-unique wire transfer subjects, and having assigned each wire transfer a unique row ID/offset.
- The exchange correctly listing all outgoing wire transfers including having the appropriate justifications (aggregation or reserve closure) for the respective amounts and target accounts.
- Wire transfers that the exchange has failed to execute that were due. Note that small delays here can be normal as wire transfers may be in flight.

3.4.3 Auditor's HTTP Service

The auditor exposes a web server with the auditor-httpd process. Currently, it shows a website that allows the customer to add the auditor to the list of trusted auditors in their wallet. It also exposes an endpoint for merchants to submit deposit confirmations. These merchant-submitted deposit confirmations are checked against the deposit permissions in the exchange's database to detect compromised signing keys or missing writes, as described in Section ??.

In the future, we plan for the auditor to expose additional endpoints where wallets and merchant backends can submit (cryptographic) proofs of misbehavior from an exchange. The goal would be to automatically verify the proofs, take corrective action by including the information in the audit report, and possibly even compensating the victim.

3.5 Merchant Backend

The merchant backend is a component that abstracts away the details of processing payments and provides a simple HTTP API. The merchant backend handles cryptographic operations (signature verification, signing), secret management, and communication with the exchange.

The backend API is divided into two types of HTTP endpoints:

1. Functionality that is accessed internally by the merchant. These APIs typically require authentication and/or are only accessible from within the private network of the merchant.

2. Functionality that is exposed publicly on the Internet and accessed by the customer's wallet and browser.

   A typical merchant has a storefront component that customers visit with their browser, as well as a back office component that allows the merchant to view information about payments that customers made and that integrates with other components such as order processing and shipping.

   Processing Payments

   To process a payment, the storefront first instructs the backend to create an order. The order contains information relevant to the purchase and is, in fact, a subset of the information contained in the contract terms. The backend automatically adds missing information to the order details provided by the storefront. The full contract terms can only be signed once the customer provides the claim public key for the contract.

   Each order is uniquely identified by an order ID, which can be chosen by the storefront or automatically generated by the backend. The order ID can be used to query the status of the payment. If the customer did not pay for an order ID yet, the response from the backend includes a payment redirect URL. The storefront can redirect the customer to this payment redirect URL; visiting the URL will trigger the customer's browser/wallet to prompt for a payment.

   To simplify the implementation of the storefront, the merchant backend can serve a page to the customer's browser that triggers the payment via the HTTP 402 status code and the corresponding headers and provides a fallback (in the form of a `:pay` link) for loosely integrated browsers. When checking the status of a payment that is not settled yet, the response from the merchant backend will contain a payment redirect URL. The storefront redirects the browser to this URL, which is served by the merchant backend and triggers the payment.

The code snippet shown in Figure 4.10 implements the core functionality of a merchant frontend that prompts the customer for a donation (upon visiting `/donate` with the right query parameters) and shows a donation receipt on the fulfillment page with URL `/receipt`. The code snippet is written in Python and uses the Flask library to process HTTP requests. The helper functions `backend_post` and `backend_get` make an HTTP POST/GET request to the merchant backend, respectively, with the given request body/query parameters.

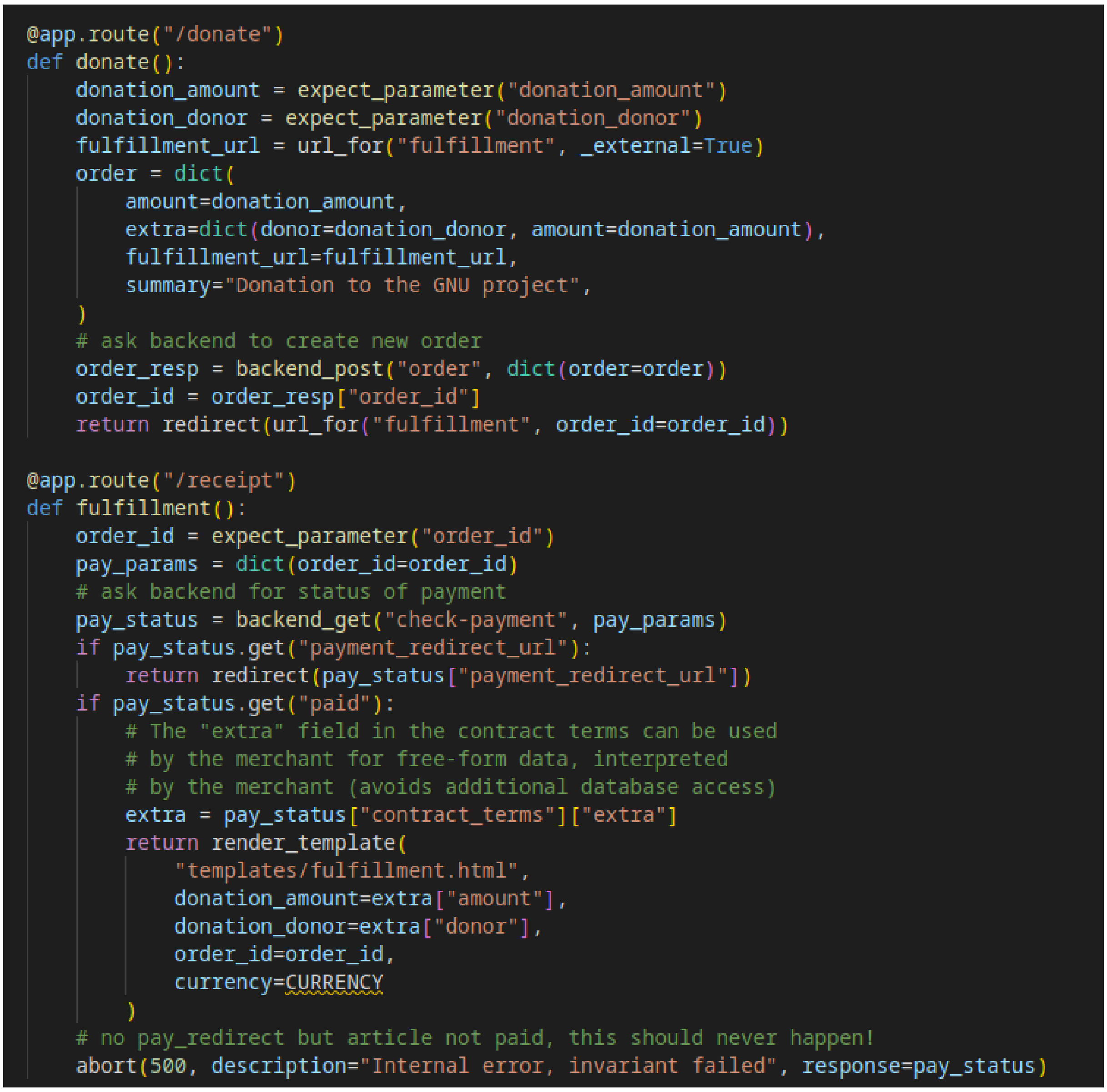
3.5.1 Back Office APIs

The back office API allows the merchant to query information about the history and status of payments, as well as correlate wire transfers to the merchant's bank account with the respective our model payment. This API is necessary to allow integration with other parts of the merchant's e-commerce infrastructure.

Example Merchant Frontends

We implemented the following applications using the merchant backend API.

- **Blog Merchant:** The blog merchant's landing page has a list of article titles with a teaser. When following the link to the article, the customer is asked to pay to view the article.
- **Donations:** The donations frontend allows the customer to select a project to donate to. The fulfillment page shows a donation receipt.
- **Codeless Payments:** The codeless payment frontend is a prototype for a user interface that allows merchants to sell products on their website without having to write code to integrate with the merchant backend. Instead, the merchant uses a web interface to manage products and their available stock. The codeless payment frontend then creates an HTML snippet with a payment button that the merchant can copy-and-paste integrate into their storefront.
- **Survey** The survey frontend showcases the tipping functionality of Our Model. The user fills out a survey and receives a tip for completing it.
- **Back office** The example back-office application shows the history and status of payments processed by the merchant.



```python
@app.route("/donate")
def donate():
    donation_amount = expect_parameter("donation_amount")
    donation_donor = expect_parameter("donation_donor")
    fulfillment_url = url_for("fulfillment", _external=True)
    order = dict(
        amount=donation_amount,
        extra=dict(donor=donation_donor, amount=donation_amount),
        fulfillment_url=fulfillment_url,
        summary="Donation to the GNU project",
    )
    # ask backend to create new order
    order_resp = backend_post("order", dict(order=order))
    order_id = order_resp["order_id"]
    return redirect(url_for("fulfillment", order_id=order_id))

@app.route("/receipt")
def fulfillment():
    order_id = expect_parameter("order_id")
    pay_params = dict(order_id=order_id)
    # ask backend for status of payment
    pay_status = backend_get("check-payment", pay_params)
    if pay_status.get("payment_redirect_url"):
        return redirect(pay_status["payment_redirect_url"])
    if pay_status.get("paid"):
        # The "extra" field in the contract terms can be used
        # by the merchant for free-form data, interpreted
        # by the merchant (avoids additional database access)
        extra = pay_status["contract_terms"]["extra"]
        return render_template(
            "templates/fulfillment.html",
            donation_amount=extra["amount"],
            donation_donor=extra["donor"],
            order_id=order_id,
            currency=CURRENCY
        )
    # no pay_redirect but article not paid, this should never happen!
    abort(500, description="Internal error, invariant failed", response=pay_status)
```

Figure 3.9: Code snippet with core funA denomination's lifetime.ctionality of a merchant frontend to accept donations.

3.5.2 Wallet

The wallet manages the customer's reserves and coins, letting the customer view and pay for contracts from merchants. It can be seen in operation in Section 1.3.

The reference implementation of the wallet is written in the TypeScript language against the WebExtension API[6], a cross-browser mechanism for browser extensions. The reference wallet is a "tightly integrated" wallet, as it directly hooks into the browser to process responses with the HTTP status code "402 Payment Required".

[6]A cross-browser mechanism for browser extensions.

Many cryptographic operations needed to implement the wallet are not commonly available in a browser environment. We cross-compile the utility library written in C as well as its dependencies (such as libgcrypt) to asm.js (and WebAssembly on supported platforms) using the LLVM-based emscripten toolchain[7].

Cryptographic operations run in an isolated process implemented as a WebWorker[8]. This design allows the relatively slow cryptographic operations to run concurrently in the background in multiple threads. Since the crypto WebWorkers are started on-demand, the wallet only uses minimal resources when not actively used.

3.5.3 Optimizations

To improve the perceived performance of cryptographic operations, the wallet optimistically creates signatures in the background while the user is looking at the "confirm payment" dialog. If the user does not accept the contract, these signatures are thrown away instead of being sent to the merchant. This effectively hides the latency of the most expensive cryptographic operations, as they are done while the user consciously needs to make a decision on whether to proceed with a payment.

3.5.4 Coin Selection

The wallet hides the implementation details of fractionally spending different denominations from the user and automatically selects which denominations to use for withdrawing a given amount, as well as which existing coins to (partially) spend for a payment. Denominations for withdrawal are greedily selected, starting with the largest denomination that fits into the remaining amount to withdraw. Coin selection for spending proceeds similarly, but first checks if there is a single coin that can be partially spent to cover the whole amount. After each payment, the wallet automatically refreshes coins with a remaining amount large enough to be refreshed. We discuss a simple simulation of the current coin selection algorithm in Section 4.8.2.

A more advanced coin selection would also consider the fee structure of the exchange, minimizing the number of coins as well as the fees incurred by the various operations. The wallet could additionally learn typical amounts that the user spends and adjust withdrawn denominations accordingly to further minimize costs. An opposing concern to the financial cost is the anonymity of customers, which is improved when the spending behavior of wallets is as similar as possible.

[7]See Zak11 for reference.

[8]See reference for WebWorkers.

3.5.5 Wallet Detection

When websites such as merchants or banks try to signal the wallet—for example, to request a payment or trigger reserve creation—it is possible that the customer simply has no wallet installed. To accommodate this situation in a user-friendly way, the HTTP response containing signaling to the wallet should contain as a response body an HTML page with (1) a link to manually open loosely integrated wallets and (2) instructions on how to install a wallet if the user does not already have one.

It might seem useful to dynamically update page content depending on whether the wallet is installed, for example, to hide or show a "Pay with wallet" or "Withdraw to wallet" option. This functionality cannot be provided in general, as only the definitive presence of a wallet can be detected, but not its absence when the wallet is only loosely integrated in the user's browser via a handler for the URI scheme.

We nevertheless consider the ability to know whether a customer has definitely installed a wallet useful (if only for the user to confirm that the installation was successful) and expose two APIs to query this. The first one is JavaScript-based and allows registering a callback for when the presence/absence of the wallet is detected. The second method works without any JavaScript on the merchant's page and uses CSS[9] to dynamically show/hide elements on the page marked with the special `installed-show` and `installed-hide` CSS classes, whose visibility is changed when a wallet browser extension is loaded.

Browser fingerprinting[10] is a concern with any additional APIs made available to websites, either by the browser itself or by browser extensions. Since a website can simply try to trigger a payment to determine whether a tightly integrated wallet is installed, one bit of additional fingerprinting information is already available through the usage of the wallet. The dynamic detection methods do not, however, expose any information that is not already available to websites by signaling the wallet through HTTP headers.

3.5.6 Wallet Signaling

We now define more precisely the algorithm that the wallet executes when a website signals to that wallet that an operation related to payments should be triggered, either by opening a `pay` URL or by responding with HTTP 402 and at least one specific header.

The steps to process a payment trigger are as follows. The algorithm takes the following parameters:

- `current_url` (the URL of the page that raises the 402 status or null if triggered by

[9] See CSS11 for reference.
[10] See Mul+13 for reference.

a `pay` URL)

- `contract_url`
- `resource_url`
- `session_id`
- `offer_url`
- `refund_url`
- `tip_token` (from the "payment..." headers or `pay` URL parameters respectively)

1. If `resource_url` is a non-empty string, set `target_url` to `resource_url`; otherwise, set `target_url` to `current_url`.
2. If `target_url` is empty, stop.
3. If there is an existing payment *p* whose fulfillment URL equals `target_url` and either `current_url` is null or `current_url` has the same origin as either the fulfillment URL or payment URL in the contract terms, then:
   (a) If `session_id` is non-empty and the last session ID for payment *p* was recorded in the wallet with session signature `sig`, construct a fulfillment instance URL from `sig` and the order ID of *p*.
   (b) Otherwise, construct an extended fulfillment URL from the order ID of *p*.
   (c) Navigate to the extended fulfillment URL constructed in the previous step and stop.
4. If `contract_url` is a non-empty URL, execute the steps for processing a contract URL (with `session_id`) and stop.
5. If `offer_url` is a non-empty URL, navigate to it and stop.
6. If `refund_url` is a non-empty URL, process the refund and stop.
7. If `tip_url` is a non-empty URL, process the tip and stop.

For interactive web applications that request payments, such as games or single page apps (SPAs), the payments initiated by navigating to a page with HTTP status code 402 are not appropriate, since the state of the web application is destroyed by the navigation. Instead, the wallet can offer a JavaScript-based API, exposed as a single function with a subset of the parameters of the 402-based payment: `contract_url`, `resource_url`, `session_id`, `refund_url`, `offer_url`, `tip_token`. Instead of navigating away, the result of the operation is returned as a JavaScript promise (either a payment receipt, refund confirmation, tip success status, or error). If user input is required (e.g., to ask for a confirmation for a payment), the page's status must not be destroyed. Instead, an overlay or separate tab/window displays the prompt to the user.

3.6 Cryptographic Protocols

In this section, we describe the main cryptographic protocols in more detail. The more abstract, high-level protocols from Section 3.5.1 are instantiated and embedded in concrete protocol diagrams that can hopefully serve as a reference for implementors.

For ease of presentation, we do not provide a bit-level description of the cryptographic protocol. Some details from the implementation are left out, such as fees, additional timestamps in messages, and checks for the expiration of denominations. Furthermore, we do not specify the exact responses in the error cases, which in the actual implementation should include signatures that could be used during a legal dispute. Similarly, the actual implementation contains some additional signatures on messages sent that allow proving to a third party that a participant did not follow the protocol.

As we are dealing with financial transactions, we explicitly describe whenever entities need to safely write data to persistent storage. As long as the data persists, the protocol can be safely resumed at any step. Persisting data is cumulative; that is, an additional persist operation does not erase the previously stored information.

The implementation also records additional entries in the exchange's database that are needed for auditing.

3.6.1 Preliminaries

In our protocol definitions, we write `check COND` to abort the protocol with an error if the condition `COND` is false.

We use the following algorithms:

- `Ed25519.Keygen()` $\to \langle sk, pk \rangle$ to generate an Ed25519 key pair.
- `Ed25519.GetPub(sk)` $\to pk$ to derive the public key from an Ed25519 secret key.
- `Ed25519.Sign(sk, m)` $\to \sigma$ to create a signature σ on message m using secret key sk.
- `Ed25519.Verify(pk, `σ`, m)` $\to b$ to check if σ is a valid signature from pk on message m.

- `HKDF(n, k, s)` $\rightarrow m$ is the HMAC-based key derivation function [KE10], producing an output m of n bits from the input key material k and salt s.

We write Z_N^* for the multiplicative group of integers modulo N. Given an $r \in Z_N^*$, we write r^{-1} for the multiplicative inverse modulo N of r.

We write $H(m)$ for the SHA-512 hash of a bit string.

We write $FDH(N, m)$ for the full domain hash that maps the bit string m to an element of Z_N^*. Specifically, $FDH(N, m)$ is computed by first computing $H(m)$. Let $b := \lceil \log_2 N \rceil$. The full domain hash is then computed by iteratively computing a HKDF to obtain b bits of output until the b-bit value is below N. The inputs to the HKDF are a "secret key", a fixed context plus a 16-bit counter (in big endian) as a context chunk that is incremented until the computation succeeds. For the source key material, we use a binary encoding of the public RSA key with modulus N.

Here, the public RSA key is encoded by first expressing the number of bits of the modulus and the public exponent as 16-bit numbers in big endian, followed by the two numbers (again in unsigned big endian encoding). For the context, the C-string "RSA-FDA FTpsW!" (without 0-termination) is used. For the KDF, we instantiate the HKDF described in RFC 5869 [KE10] using HMAC-SHA512 as XTR and HMAC-SHA256 as PRF*. Let the result of the first successful iteration of the HKDF function be r with $0 \leq r < N$. Then, to protect against a malicious exchange when blinding values, the $FDH(N, m)$ function checks that $\gcd(r, n) = 1$. If not, the $FDH(n, m)$ calculation fails because n is determined to be malicious.

The expression $x \leftarrow X$ denotes uniform random selection of an element x from set X. We use $\text{SelectSeeded}(s, X) \rightarrow x$ for pseudo-random uniform selection of an element x from set X and seed s. Here, the result is deterministic for fixed inputs s and X.

The exchange's denomination signing key pairs $\{\langle sk_{Di}, pk_{Di} \rangle\}$ are RSA key pairs, and thus $pk_{Di} = \langle e_i, N_i \rangle$, $sk_{Di} = d_i$. We write $D(pk_{Di})$ for the financial value of the denomination pk_{Di}.

3.6.2 Withdrawing

The withdrawal protocol is defined in Figure 3.10. The following additional algorithms are used, which we only define informally here:

- `CreateBalance(Wp, v)` $\rightarrow \perp$ is used by the exchange, and has the side-effect of creating a reserve record with balance v and reserve public key (effectively the identifier of the reserve) W_p.

- `GetWithdrawR()` $\rightarrow \{\perp, _C\}$ is used by the exchange, and checks if there is an existing withdraw request ρ. If the existing request exists, the existing blind signature σ_C over coin C is returned. On a fresh request, $\perp$ is returned.

- `BalanceSufficient(W`$_s, pk_{D_t}$`)`$\rightarrow b$ `is used by the exchange, and returns true if the balance in the reserve identified by` W_s `is sufficient to withdraw at least one coin of denomination` pk_{D_t}`.`

- `DecreaseBalance(W`$_s, pk_{D_t}$`)`$\rightarrow\perp$ `is used by the exchange, and decreases the amount left in the reserve identified by` W_s `by the amount` $D(pk_{D_t})$ `that the denomination` pk_{D_t} `represents.`

3.6.3 Payment Transactions

The payment protocol is defined in two parts. First, the spend protocol in Figure 3.11 defines the interaction between a merchant and a customer. The customer obtains the contract terms (as ρ_P) from the merchant, and sends the merchant deposit permissions as a payment. The deposit protocol in Figure 3.12 defines how subsequently the merchant sends the deposit permissions to the exchange to detect double-spending and ultimately to receive a bank transfer from the exchange.

Note that in practice the customer can also execute the deposit protocol on behalf of the merchant. This is useful in situations where the customer has network connectivity but the merchant does not. It also allows the customer to complete a payment before the payment deadline if a merchant unexpectedly becomes unresponsive, allowing the customer to later prove that they paid on time.

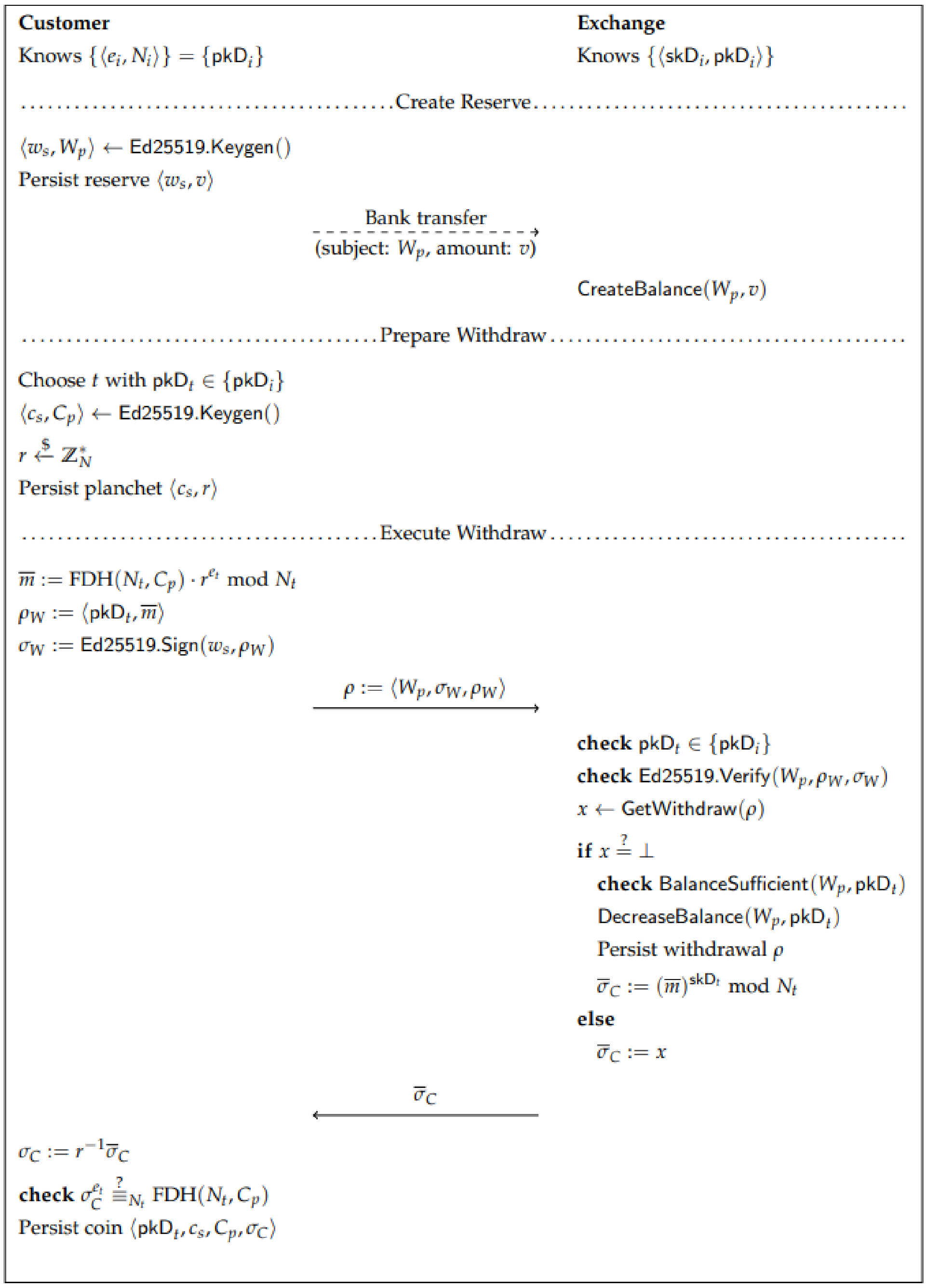
We limit the description to one exchange here, but in practice, the merchant communicates to the customer the exchanges that it supports, in addition to the account information A_M that might differ between exchanges.

We use the following algorithms, defined informally here:

- **SelectPayCoins**(v, EM) $\rightarrow \{\langle \text{coin}_i, f_i \rangle\}$ selects fresh coins (signed with denomination keys from exchange EM) to pay for the amount v. The result is a set of coins together with the fraction of each coin that must be spent such that the amounts contributed by all coins sum up to v.

- **MarkDirty**(coin, f) $\rightarrow \perp$ subtracts the fraction f from the available amount left on a coin, and marks the coin as dirty (to trigger refreshing in case f is below the denomination value). Thus, assuming the coin has any residual value, the customer's wallet will do a refresh on the coin and not use it for further payments. This provides unlinkability of transactions made with change arising from paying with fractions of a coin's denomination.

- **Deposit**(EM, D_i) $\rightarrow b$ executes the second part of the payment protocol (i.e., the deposit) with exchange EM, using deposit permission D_i.
- **GetDeposit**(C_p, h) $\rightarrow \{\perp, \rho(D, i)\}$ checks in the exchange's database for an existing processed deposit permission on coin C_p for the contract identified by h. The algorithm returns the existing deposit permission $\rho(D, i)$, or $\perp$ if a matching deposit permission is not recorded.
- **IsOverspending**(C_p, pk_D, f) $\rightarrow b$ checks in the exchange's database if at least the fraction f of the coin C_p of denomination pk_D is still available for use, based on existing spend/withdraw records of the exchange.
- **MarkFractionalSpend**(C_p, f) $\rightarrow \perp$ adds a spending record to the exchange's database, indicating that fraction f of coin C_p has been spent (in addition to existing spending/refreshing records).
- **ScheduleBankTransfer**(A_M, f, pk_D, h_c) $\rightarrow \perp$ schedules a bank transfer from the exchange to the account identified by A_M, for subject h_c and for the amount $f \cdot D(pk_D)$.



Customer

Knows $\{\langle e_i, N_i \rangle\} = \{\mathsf{pkD}_i\}$

Exchange

Knows $\{\langle \mathsf{skD}_i, \mathsf{pkD}_i \rangle\}$

Create Reserve

$\langle w_s, W_p \rangle \leftarrow \mathsf{Ed25519.Keygen}()$

Persist reserve $\langle w_s, v \rangle$

$\xrightarrow[\text{(subject: } W_p\text{, amount: } v\text{)}]{\text{Bank transfer}}$

$\mathsf{CreateBalance}(W_p, v)$

Prepare Withdraw

Choose t with $\mathsf{pkD}_t \in \{\mathsf{pkD}_i\}$

$\langle c_s, C_p \rangle \leftarrow \mathsf{Ed25519.Keygen}()$

$r \xleftarrow{\$} \mathbb{Z}_N^*$

Persist planchet $\langle c_s, r \rangle$

Execute Withdraw

$\overline{m} := \mathsf{FDH}(N_t, C_p) \cdot r^{e_t} \bmod N_t$

$\rho_W := \langle \mathsf{pkD}_t, \overline{m} \rangle$

$\sigma_W := \mathsf{Ed25519.Sign}(w_s, \rho_W)$

$\xrightarrow{\rho := \langle W_p, \sigma_W, \rho_W \rangle}$

```
check $\mathsf{pkD}_t \in \{\mathsf{pkD}_i\}$
check $\mathsf{Ed25519.Verify}(W_p, \rho_W, \sigma_W)$
$x \leftarrow \mathsf{GetWithdraw}(\rho)$
if $x \overset{?}{=} \bot$
  check $\mathsf{BalanceSufficient}(W_p, \mathsf{pkD}_t)$
  $\mathsf{DecreaseBalance}(W_p, \mathsf{pkD}_t)$
  Persist withdrawal $\rho$
  $\overline{\sigma}_C := (\overline{m})^{\mathsf{skD}_t} \bmod N_t$
else
  $\overline{\sigma}_C := x$
```

$\xleftarrow{\overline{\sigma}_C}$

$\sigma_C := r^{-1}\overline{\sigma}_C$

check $\sigma_C^{e_t} \overset{?}{\equiv}_{N_t} \mathsf{FDH}(N_t, C_p)$

Persist coin $\langle \mathsf{pkD}_t, c_s, C_p, \sigma_C \rangle$

Figure 3.10: Withdrawal protocol diagram.

3.6.4 Refreshing and Linking

The refresh protocol is defined in Figures 3.14 and 3.15 . The refresh protocol allows the customer to obtain change for the remaining fraction of the value of a coin. The change is generated as a fresh coin that is unlinkable to the dirty coin to anyone except for the owner of the dirty coin. A naïve implementation of a refresh protocol that just gives the customer a new coin could be used for peer-to-peer transactions that hide income from tax authorities. Thus, with probability $(1 - \frac{1}{\kappa})$, the refresh protocol records information that allows the owner of the original coin to obtain the refreshed coin from the original coin via the linking protocol (illustrated in Figure 3.16).

We use the following algorithms, defined informally here:

- **RefreshDerive** is defined in Figure 4.15.
- **GetOldRefresh**$(\rho_{RC}) \rightarrow \{\perp, \gamma\}$ returns the past choice of γ if ρ_{RC} is a refresh commit message that has been seen before, and $\perp$ otherwise.
- **IsConsistentChallenge**$(\rho_{RC},\ \gamma) \rightarrow \{\perp, \top\}$ returns $\top$ if no refresh-challenge has been persisted for the refresh operation by commitment ρ_{RC} or if γ is consistent with the persisted (and thus previously received) challenge; returns $\perp$ otherwise.
- **LookupLink**$(C_p) \rightarrow \{\langle \rho_L^{(i)}, \sigma_L^{(i)}, \sigma_C^{(i)} \rangle\}$ looks up refresh records on the coin with public key C_p in the exchange's database and returns the linking message $\rho_L^{(i)}$, linking signature $\sigma_L^{(i)}$, and blinded signature $\sigma_C^{(i)}$ for each refresh record i.

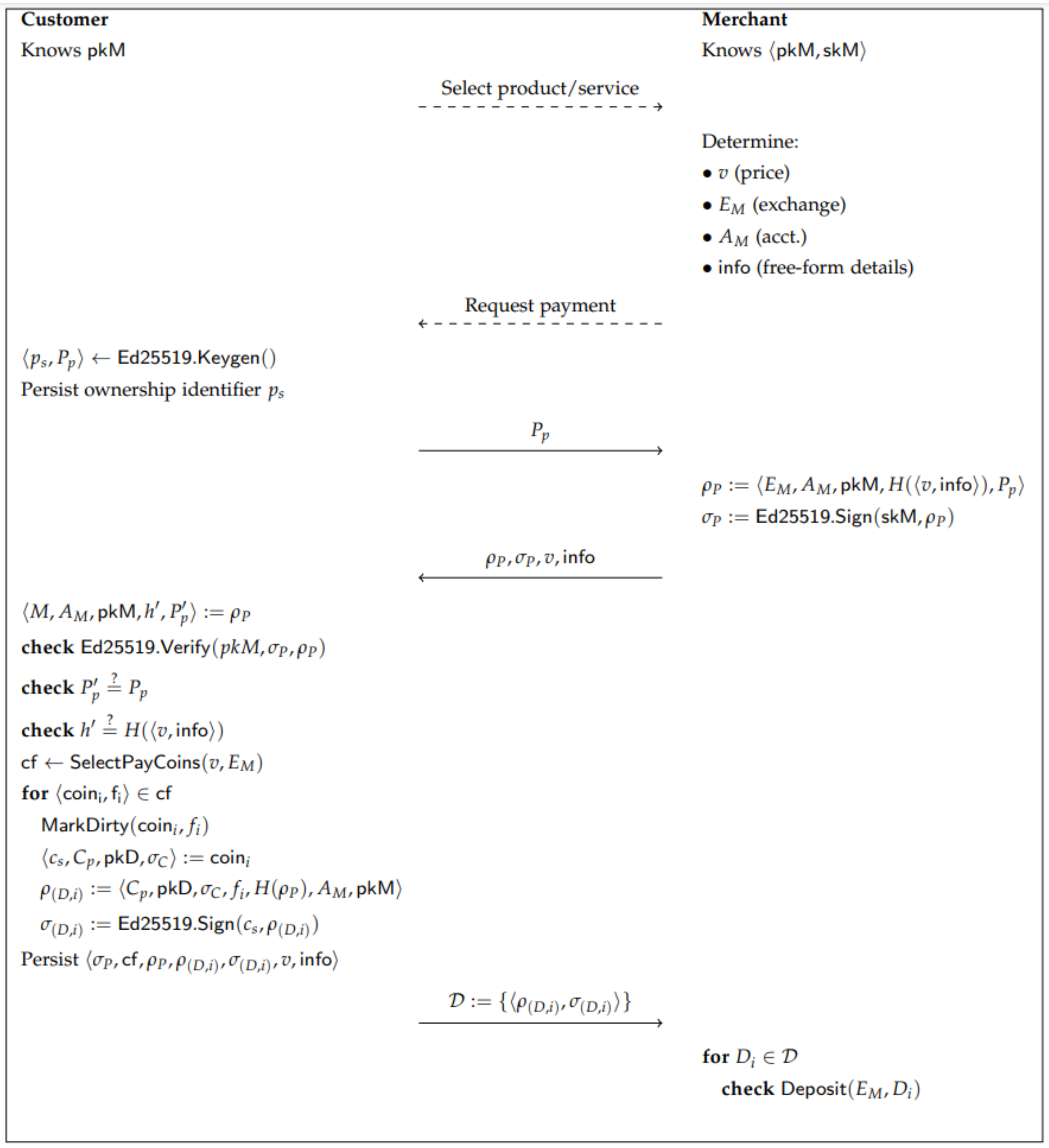
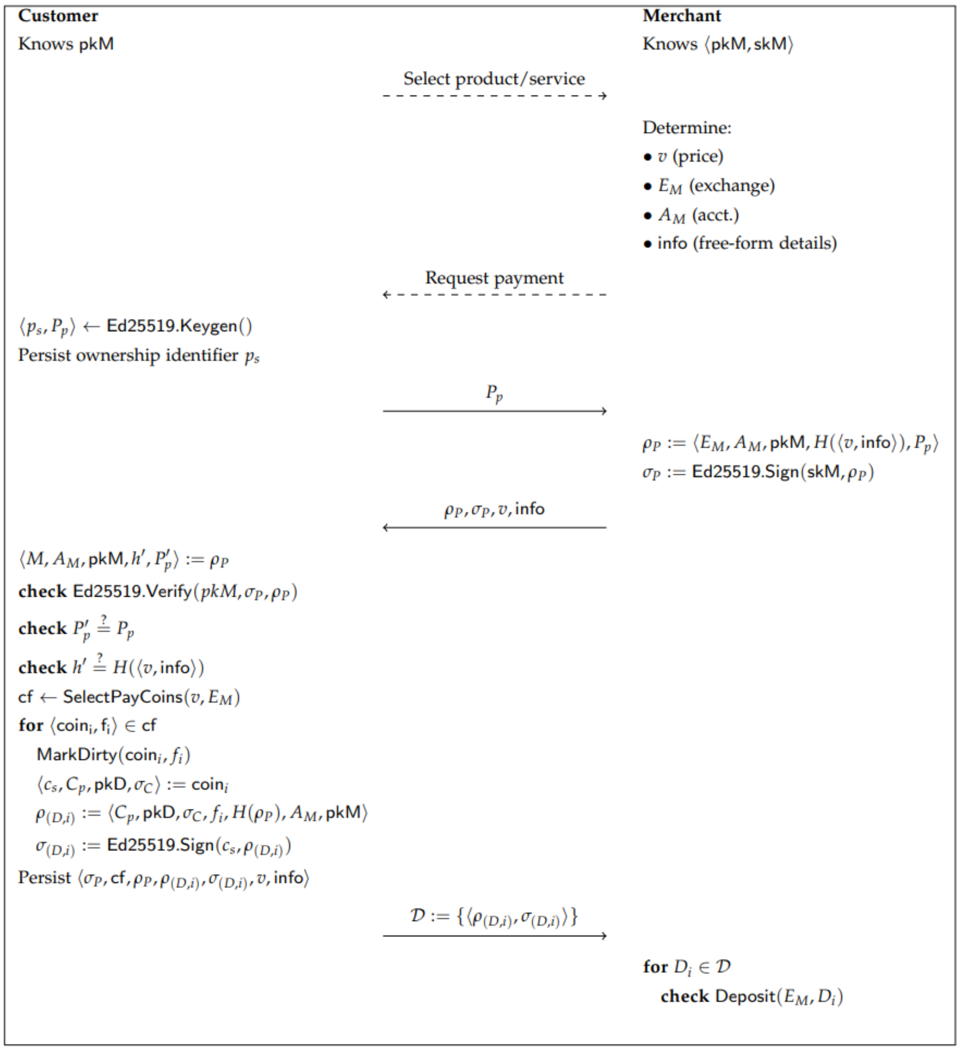

Customer
Knows pkM
Merchant
Knows $\langle \mathsf{pkM}, \mathsf{skM} \rangle$
Select product/service
Determine:
• v (price)
• E_M (exchange)
• A_M (acct.)
• info (free-form details)
Request payment
$\langle p_s, P_p \rangle \leftarrow \mathsf{Ed25519.Keygen}()$
Persist ownership identifier p_s
P_p
$\rho_P := \langle E_M, A_M, \mathsf{pkM}, H(\langle v, \mathsf{info} \rangle), P_p \rangle$
$\sigma_P := \mathsf{Ed25519.Sign}(\mathsf{skM}, \rho_P)$
$\rho_P, \sigma_P, v, \mathsf{info}$
$\langle M, A_M, \mathsf{pkM}, h', P'_p \rangle := \rho_P$
check $\mathsf{Ed25519.Verify}(pkM, \sigma_P, \rho_P)$
check $P'_p \overset{?}{=} P_p$
check $h' \overset{?}{=} H(\langle v, \mathsf{info} \rangle)$
$\mathsf{cf} \leftarrow \mathsf{SelectPayCoins}(v, E_M)$
for $\langle \mathsf{coin_i}, \mathsf{f_i} \rangle \in \mathsf{cf}$
$\mathsf{MarkDirty}(\mathsf{coin}_i, f_i)$
$\langle c_s, C_p, \mathsf{pkD}, \sigma_C \rangle := \mathsf{coin}_i$
$\rho_{(D,i)} := \langle C_p, \mathsf{pkD}, \sigma_C, f_i, H(\rho_P), A_M, \mathsf{pkM} \rangle$
$\sigma_{(D,i)} := \mathsf{Ed25519.Sign}(c_s, \rho_{(D,i)})$
Persist $\langle \sigma_P, \mathsf{cf}, \rho_P, \rho_{(D,i)}, \sigma_{(D,i)}, v, \mathsf{info} \rangle$
$\mathcal{D} := \{\langle \rho_{(D,i)}, \sigma_{(D,i)} \rangle\}$
for $D_i \in \mathcal{D}$
check $\mathsf{Deposit}(E_M, D_i)$

Figure 3.11: The spend protocol is executed between the customer and the merchant for the purchase of an article of price v using coins from exchange EM. The merchant has provided his account details to the exchange under an identifier AM. The customer can identify themselves as the one who received the offer using ps.



Customer

Knows pkM

$\langle p_s, P_p \rangle \leftarrow$ Ed25519.Keygen()

Persist ownership identifier p_s

$\langle M, A_M, \text{pkM}, h', P'_p \rangle := \rho_P$

check Ed25519.Verify(pkM, σ_P, ρ_P)

check $P'_p \overset{?}{=} P_p$

check $h' \overset{?}{=} H(\langle v, \text{info} \rangle)$

```
cf ← SelectPayCoins($v, E_M$)
for ⟨coin_i, f_i⟩ ∈ cf
    MarkDirty(coin_i, $f_i$)
    $\langle c_s, C_p, \text{pkD}, \sigma_C \rangle := \text{coin}_i$
    $\rho_{(D,i)} := \langle C_p, \text{pkD}, \sigma_C, f_i, H(\rho_P), A_M, \text{pkM} \rangle$
    $\sigma_{(D,i)} :=$ Ed25519.Sign$(c_s, \rho_{(D,i)})$
Persist $\langle \sigma_P, \text{cf}, \rho_P, \rho_{(D,i)}, \sigma_{(D,i)}, v, \text{info} \rangle$
```

Merchant

Knows $\langle \text{pkM}, \text{skM} \rangle$

Determine:

- v (price)
- E_M (exchange)
- A_M (acct.)
- info (free-form details)

$\rho_P := \langle E_M, A_M, \text{pkM}, H(\langle v, \text{info} \rangle), P_p \rangle$

$\sigma_P :=$ Ed25519.Sign(skM, ρ_P)

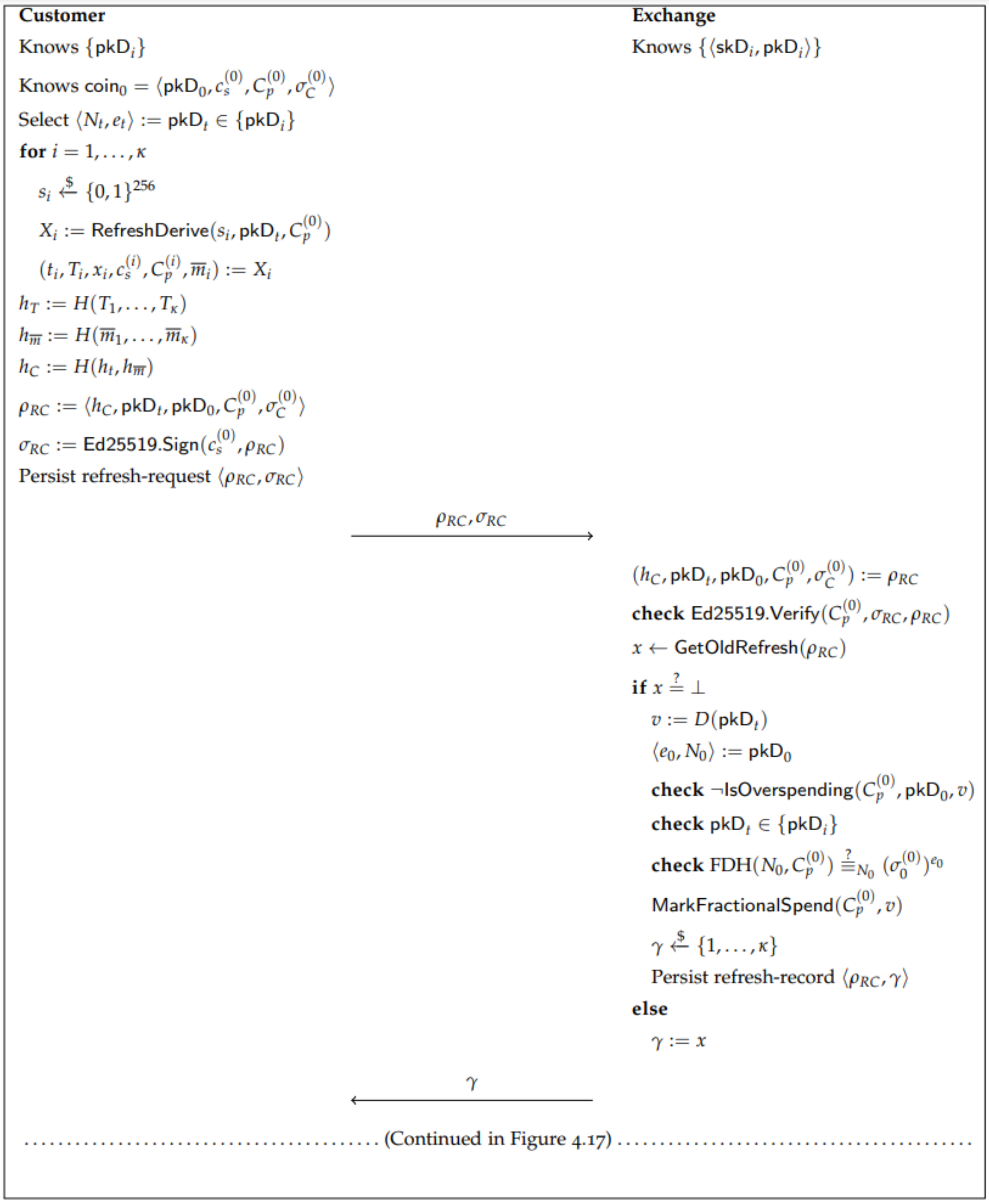
```
for $D_i \in \mathcal{D}$
    check Deposit($E_M, D_i$)
```

Select product/service
Request payment
P_p
$\rho_P, \sigma_P, v, \text{info}$
$\mathcal{D} := \{\langle \rho_{(D,i)}, \sigma_{(D,i)} \rangle\}$

Figure 3.12: The deposit protocol is run for each deposited coin $D_i \in D$ with the exchange that signed the coin.

```
RefreshDerive$(s, \langle e, N\rangle, C_p)$
$t :=$ HKDF$(256, s,$ "t"$)$
$T :=$ Curve25519.GetPub$(t)$
$x :=$ ECDH-EC$(t, C_p)$
$r :=$ SelectSeeded$(x, \mathbb{Z}^*_N)$
$c_s :=$ HKDF$(256, x,$ "c"$)$
$C_p :=$ Ed25519.GetPub$(c_s)$
$\overline{m} := r^e \cdot C_p \mod N$
return $\langle t, T, x, c_s, C_p, \overline{m}\rangle$
```

Figure 3.13: The RefreshDerive algorithm running with the seed s on dirty coin Cp to generate a fresh coin to be later signed with denomination key pkD := ⟨e, N⟩.



```
Customer
Knows $\{\mathsf{pkD}_i\}$
Knows $\mathsf{coin}_0 = \langle \mathsf{pkD}_0, c_s^{(0)}, C_p^{(0)}, \sigma_C^{(0)} \rangle$
Select $\langle N_t, e_t \rangle := \mathsf{pkD}_t \in \{\mathsf{pkD}_i\}$
for $i = 1, \ldots, \kappa$
    $s_i \xleftarrow{\$} \{0,1\}^{256}$
    $X_i := \mathsf{RefreshDerive}(s_i, \mathsf{pkD}_t, C_p^{(0)})$
    $(t_i, T_i, x_i, c_s^{(i)}, C_p^{(i)}, \overline{m}_i) := X_i$
$h_T := H(T_1, \ldots, T_\kappa)$
$h_{\overline{m}} := H(\overline{m}_1, \ldots, \overline{m}_\kappa)$
$h_C := H(h_t, h_{\overline{m}})$
$\rho_{RC} := \langle h_C, \mathsf{pkD}_t, \mathsf{pkD}_0, C_p^{(0)}, \sigma_C^{(0)} \rangle$
$\sigma_{RC} := \mathsf{Ed25519.Sign}(c_s^{(0)}, \rho_{RC})$
Persist refresh-request $\langle \rho_{RC}, \sigma_{RC} \rangle$

                    $\xrightarrow{\rho_{RC}, \sigma_{RC}}$

Exchange
Knows $\{\langle \mathsf{skD}_i, \mathsf{pkD}_i \rangle\}$
$(h_C, \mathsf{pkD}_t, \mathsf{pkD}_0, C_p^{(0)}, \sigma_C^{(0)}) := \rho_{RC}$
check $\mathsf{Ed25519.Verify}(C_p^{(0)}, \sigma_{RC}, \rho_{RC})$
$x \leftarrow \mathsf{GetOldRefresh}(\rho_{RC})$
if $x \overset{?}{=} \perp$
    $v := D(\mathsf{pkD}_t)$
    $\langle e_0, N_0 \rangle := \mathsf{pkD}_0$
    check $\neg\mathsf{IsOverspending}(C_p^{(0)}, \mathsf{pkD}_0, v)$
    check $\mathsf{pkD}_t \in \{\mathsf{pkD}_i\}$
    check $\mathrm{FDH}(N_0, C_p^{(0)}) \overset{?}{\equiv}_{N_0} (\sigma_0^{(0)})^{e_0}$
    $\mathsf{MarkFractionalSpend}(C_p^{(0)}, v)$
    $\gamma \xleftarrow{\$} \{1, \ldots, \kappa\}$
    Persist refresh-record $\langle \rho_{RC}, \gamma \rangle$
else
    $\gamma := x$

                    $\xleftarrow{\gamma}$
```

.... (Continued in Figure 4.17) ....

Figure 3.14: Refresh Protocol (Commit Phase)(also have second part look at the next page)

Customer **Exchange**

....(Continuation of 4.16)..

$\xleftarrow{\quad\quad\quad\gamma\quad\quad\quad}$

check $\mathsf{IsConsistentChallenge}(\rho_{RC}, \gamma)$

Persist refresh-challenge $\langle \rho_{RC}, \gamma \rangle$

$S := \langle s_1, \dots, s_{\gamma-1}, s_{\gamma+1}, \dots, s_\kappa \rangle$

$\rho_L = \langle C_p^{(0)}, \mathsf{pkD}_t, T_\gamma, \overline{m}_\gamma \rangle$

$\rho_{RR} = \langle T_\gamma, \overline{m}_\gamma, S \rangle$

$\sigma_L = \mathsf{Ed25519.Sign}(c_s^{(0)}, \rho_L)$

$\xrightarrow{\quad\quad\rho_{RR}, \rho_L, \sigma_L\quad\quad}$

$\langle T'_\gamma, \overline{m}'_\gamma, S \rangle := \rho_{RR}$

$\langle s_1, \dots, s_{\gamma-1}, s_{\gamma+1}, \dots, s_\kappa \rangle) := S$

check $\mathsf{Ed25519.Verify}(C_p^{(0)}, \sigma_L, \rho_L)$

for $i = 1, \dots, \gamma-1, \gamma+1, \dots, \kappa$

  $X_i := \mathsf{RefreshDerive}(s_i, \mathsf{pkD}_t, C_p^{(0)})$

  $\langle t_i, T_i, x_i, c_s^{(i)}, C_p^{(i)}, \overline{m}_i \rangle := X_i$

$h'_T = H(T_1, \dots, T_{\gamma-1}, T'_\gamma, T_{\gamma+1}, \dots, T_\kappa)$

$h'_{\overline{m}} = H(\overline{m}_1, \dots, \overline{m}_{\gamma-1}, \overline{m}'_\gamma, \overline{m}_{\gamma+1}, \dots, \overline{m}_\kappa)$

$h'_C = H(h'_T, h'_{\overline{m}})$

check $h_C \overset{?}{=} h'_C$

$\overline{\sigma}_C^{(\gamma)} := \overline{m}^{skD_t}$

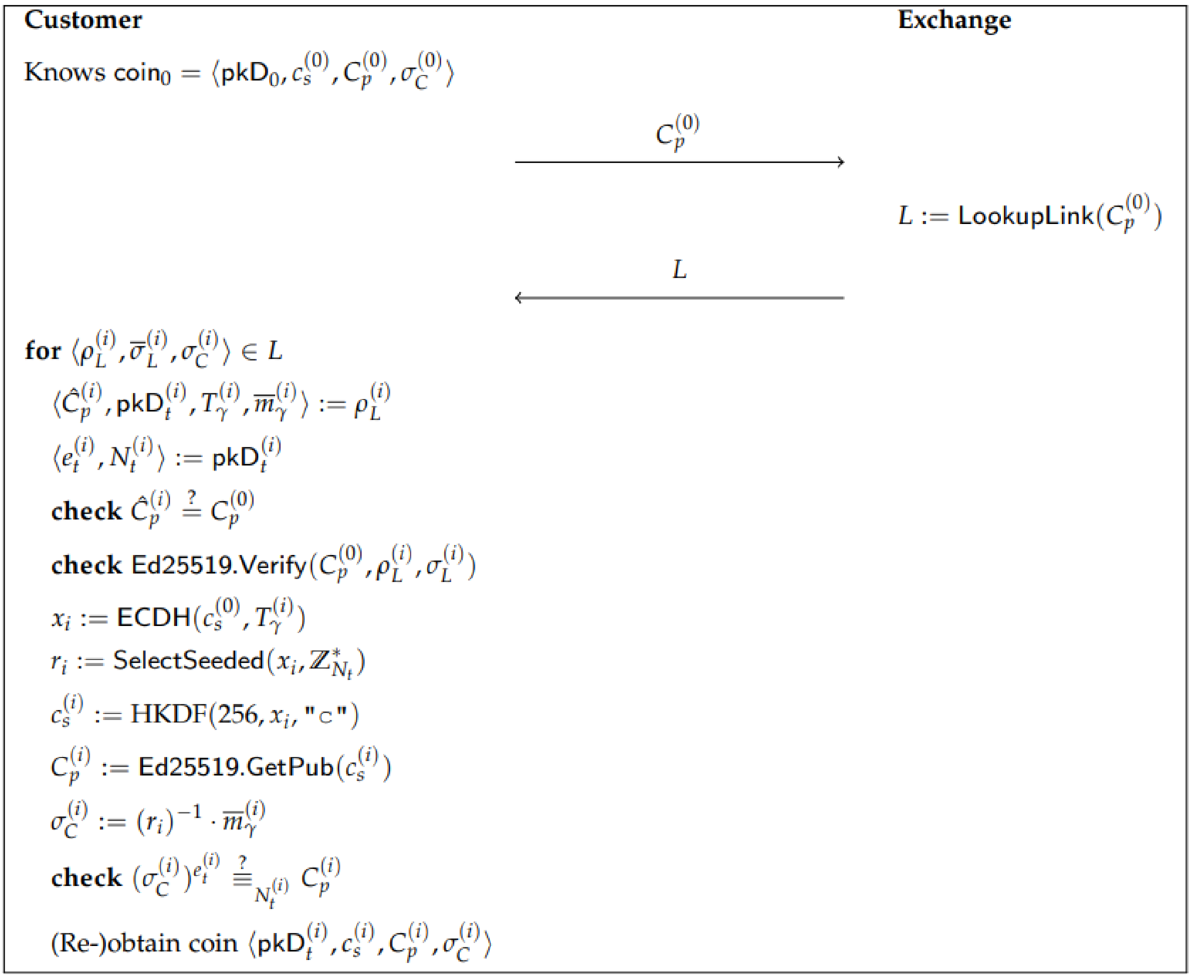
$\xleftarrow{\quad\quad\quad\overline{\sigma}_C^{(\gamma)}\quad\quad\quad}$

$\sigma_C^{(\gamma)} := r^{-1}\overline{\sigma}_C^{(\gamma)}$

check $(\sigma_C^{(\gamma)})^{e_t} \overset{?}{\equiv}_{N_t} C_p^{(\gamma)}$

Persist coin $\langle \mathsf{pkD}_t, c_s^{(\gamma)}, C_p^{(\gamma)}, \sigma_C^{(\gamma)} \rangle$

Figure 3.15: Refresh Protocol (Reveal Phase) (continued of last figure)



Customer **Exchange**

Knows $\mathsf{coin}_0 = \langle \mathsf{pkD}_0, c_s^{(0)}, C_p^{(0)}, \sigma_C^{(0)} \rangle$

$$\xrightarrow{C_p^{(0)}}$$

$L := \mathsf{LookupLink}(C_p^{(0)})$

$$\xleftarrow{L}$$

```
for $\langle \rho_L^{(i)}, \overline{\sigma}_L^{(i)}, \sigma_C^{(i)} \rangle \in L$
    $\langle \hat{C}_p^{(i)}, \mathsf{pkD}_t^{(i)}, T_\gamma^{(i)}, \overline{m}_\gamma^{(i)} \rangle := \rho_L^{(i)}$
    $\langle e_t^{(i)}, N_t^{(i)} \rangle := \mathsf{pkD}_t^{(i)}$
    check $\hat{C}_p^{(i)} \overset{?}{=} C_p^{(0)}$
    check $\mathsf{Ed25519.Verify}(C_p^{(0)}, \rho_L^{(i)}, \sigma_L^{(i)})$
    $x_i := \mathsf{ECDH}(c_s^{(0)}, T_\gamma^{(i)})$
    $r_i := \mathsf{SelectSeeded}(x_i, \mathbb{Z}_{N_t}^*)$
    $c_s^{(i)} := \mathrm{HKDF}(256, x_i, \texttt{"c"})$
    $C_p^{(i)} := \mathsf{Ed25519.GetPub}(c_s^{(i)})$
    $\sigma_C^{(i)} := (r_i)^{-1} \cdot \overline{m}_\gamma^{(i)}$
    check $(\sigma_C^{(i)})^{e_t^{(i)}} \overset{?}{\equiv}_{N_t^{(i)}} C_p^{(i)}$
    (Re-)obtain coin $\langle \mathsf{pkD}_t^{(i)}, c_s^{(i)}, C_p^{(i)}, \sigma_C^{(i)} \rangle$
```

Figure 3.16: Linking Protocol.

3.6.5 Refunds

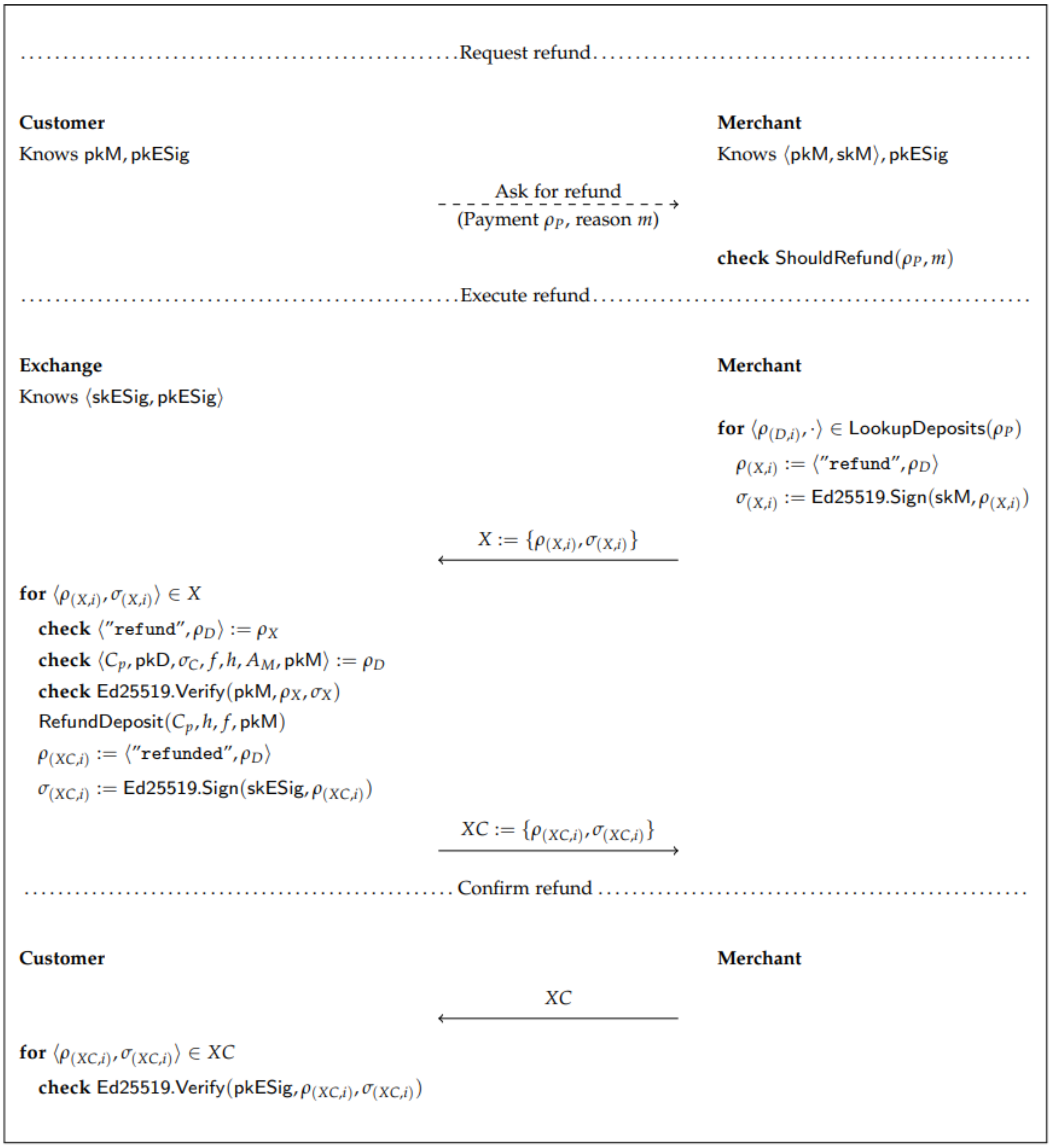
The refund protocol is defined in Figure 3.17. The customer requests from the merchant that a deposit should be "reversed," and if the merchant allows the refund, it authorizes the exchange to apply the refund and sends the refund confirmation back to the customer. Note that in practice, refunds are only possible before the refund deadline, which is not considered here.

We use the following algorithms, defined informally here:

- `ShouldRefund(P, m)` $\to \{\top, \bot\}$ is used by the merchant to check whether a refund with reason m should be given for the purchase identified by the contract terms P. The decision is made according to the merchant's business rules.

- `LookupDeposits(P, m)` $\to \{\langle (D, i), (D, i) \rangle\}$ is used by the merchant to retrieve deposit permissions that were previously sent by the customer and already deposited with the exchange.

- `RefundDeposit(Cp, h, f, pkM)` is used by the exchange to modify its database. It (partially) reverses the amount f of a deposit of coin C_p to the merchant p_kM for the contract identified by h. The procedure is idempotent, and subsequent invocations with a larger f increase the refund.



Customer: Knows pkM, pkESig

Merchant: Knows $\langle \mathsf{pkM}, \mathsf{skM} \rangle$, pkESig

Customer → Merchant: Ask for refund (Payment ρ_P, reason m)

```
Merchant:
check ShouldRefund($\rho_P, m$)
```

Exchange: Knows $\langle \mathsf{skESig}, \mathsf{pkESig} \rangle$

```
Merchant:
for $\langle \rho_{(D,i)}, \cdot \rangle \in$ LookupDeposits($\rho_P$)
    $\rho_{(X,i)} := \langle$"refund"$, \rho_D \rangle$
    $\sigma_{(X,i)} :=$ Ed25519.Sign(skM, $\rho_{(X,i)}$)
```

Merchant → Exchange: $X := \{\rho_{(X,i)}, \sigma_{(X,i)}\}$

```
Exchange:
for $\langle \rho_{(X,i)}, \sigma_{(X,i)} \rangle \in X$
    check $\langle$"refund"$, \rho_D \rangle := \rho_X$
    check $\langle C_p, \mathsf{pkD}, \sigma_C, f, h, A_M, \mathsf{pkM} \rangle := \rho_D$
    check Ed25519.Verify(pkM, $\rho_X, \sigma_X$)
    RefundDeposit($C_p, h, f$, pkM)
    $\rho_{(XC,i)} := \langle$"refunded"$, \rho_D \rangle$
    $\sigma_{(XC,i)} :=$ Ed25519.Sign(skESig, $\rho_{(XC,i)}$)
```

Exchange → Merchant: $XC := \{\rho_{(XC,i)}, \sigma_{(XC,i)}\}$

Merchant → Customer: XC

```
Customer:
for $\langle \rho_{(XC,i)}, \sigma_{(XC,i)} \rangle \in XC$
    check Ed25519.Verify(pkESig, $\rho_{(XC,i)}, \sigma_{(XC,i)}$)
```

Figure 3.17: Refund Protocol

Chapter 4

Future Developments for Model

1. **Secure Instantiations in the Standard Model**
    - Our solution currently relies on hash functions, but secure implementations in the standard model are being researched to remove the random oracle assumption. Utilizing blind signature schemes could provide secure instantiation, though it may require lesser-known assumptions.
2. **Post-Quantum and Incentive Integration**
    - With the rise of post-quantum computing, there is interest in applying these secure schemes to e-cash systems. Additionally, incorporating incentives into peer-to-peer networking platforms can enhance volunteer-driven protocols.
3. **Wallet Synchronization and Proofs**
    - Synchronization of wallets across devices is essential for user experience and must protect privacy. Methods like Private Information Retrieval (PIR) can aid in this. Furthermore, integrating machine-verified proofs will enhance the reliability of our model. We also aim to create a system for children that teaches responsible spending while ensuring privacy and preventing age-inappropriate purchases.

Chapter 5

Conclusion

This book presents our solution as an efficient protocol for value-based electronic payment systems, emphasizing security and privacy. While we believe our model to be socially and economically beneficial, it is crucial to conduct a technological impact analysis before implementing new systems that could have significant economic and socio-political consequences. Currencies fulfill three essential functions in society: they serve as a unit for measuring value, act as a medium of exchange, and function as a store of value. Evaluating how various methods perform against these requirements will be important in assessing their effectiveness and suitability for broader adoption.

Chapter 6

Bonus : Income-Transparent E-Cash: Balancing Security and Anonymity

We will formally model the security properties of our income-transparent anonymous e-cash system. We present a slightly simplified version of the implemented system (detailed in Chapter 4), articulate our desired security properties with precision, and demonstrate that our protocol instantiation effectively satisfies these properties.

6.1 Introduction to Provable Security

In developing our income-transparent anonymous e-cash system, we adopt the concept of provable security, which is a common approach for constructing formal arguments to support the security of cryptographic protocols concerning specific security properties and underlying assumptions on cryptographic primitives.

The adversary we consider is computationally bounded, meaning that the runtime is typically restricted to polynomial time relative to the security parameters (such as key length) of our protocol.

It is essential to note that a protocol labeled as "provably secure" does not guarantee security in practical implementations. Provable security results are often based on reductions, where if an effective adversary A can breach our protocol P, then it can be used to construct an adversary A' against a known secure protocol or a hard computational problem Q. The practical implications of security proofs depend on several factors:

- **Well-Studied Problems**: The complexity of the underlying problems is crucial. Some cryptographic constructs, especially those relying on pairing-based approaches, depend on intricate and exotic problems that are assumed to be hard but may not be.

- **Tightness of Reductions**: The security proof might indicate that if P can be solved in time T, then Q can be solved in time $f(T)$, where f is a function of T (e.g., T^2). This may necessitate larger key sizes or security parameters for P than for Q.

- **Assumptions in Reductions**: The use of assumptions such as the Random Oracle Model (ROM) raises additional considerations. In ROM, hash functions are treated as queries to a trusted third party that returns a random value for each unique input. Although many consider the ROM a practical assumption, it has been shown that certain protocols secure under the ROM may not be secure with any concrete hash function.

Additionally, a provably secure protocol does not inherently translate to a secure implementation, as side-channel vulnerabilities and fault injection attacks are typically not modeled in these proofs. Furthermore, the stated security properties may not be exhaustive or adequate for our application's requirements.

For our purposes, we emphasize game-based provable security rather than simulation-based provable security, as the former is more relevant to our implementation of the income-transparent anonymous e-cash system.

6.1.1 Algorithms, Oracles, and Games

To analyze the security of our income-transparent anonymous e-cash system, we begin by formally modeling the protocol and its desired security properties against an adversary with specific capabilities. This model is independent of any concrete instantiation of the protocol and is described on a syntactic level.

The possible operations of our protocol are defined abstractly as signatures of algorithms. Each algorithm will later be instantiated through a concrete implementation, formally represented as a program for a probabilistic Turing machine. For example, our system may consist of the following algorithms:

- **KeyGen(1λ)** $\rightarrow$ (sk, pk): A probabilistic algorithm that generates a fresh key pair, consisting of a secret key sk of length λ and its corresponding public key pk, upon input 1^λ.

- **Sign(sk, m)** $\rightarrow \sigma$: An algorithm that signs the bit string m using the secret key sk to produce the signature σ.

- **Verify(pk, σ, m)** $\rightarrow b$: An algorithm that determines whether σ is a valid signature on m made with the secret key corresponding to the public key pk. It outputs a flag $b \in \{0, 1\}$ to indicate whether the signature is valid (1) or invalid (0).

This abstract syntax can be instantiated with various concrete signature protocols suitable for our e-cash system.

In addition to the computational power assigned to the adversary, its capabilities are defined through oracles. Oracles function as an API provided to the adversary, enabling interaction with the protocol's environment. Unlike algorithms, access to oracles is often restricted, and oracles can maintain state that is not directly accessible to the adversary. They typically allow the adversary to access information or trigger operations within the protocol.

Formally, oracles extend the Turing machine that runs the adversary, allowing it to submit queries for interaction. For our e-cash system, an adversary might be given access to an **OSign** oracle, which generates signatures using secret keys that the adversary cannot access directly. Different definitions of **OSign** confer different capabilities to the adversary, leading to varying security properties:

- If the signing oracle **OSign(m)** is defined to return a signature σ for a given message m, the adversary gains the ability to perform chosen message attacks.
- If **OSign(·)** is defined to return a pair (σ, m) consisting of a signature σ on a random message m, the adversary's power is reduced to known message attacks.

While oracles describe potential system interactions, it is often more convenient to describe complex multi-round interactions as a special form of algorithm called an interactive protocol. This protocol takes the identifiers of the communicating parties and their (private) inputs as parameters, orchestrating the interaction among them. The adversary can then initiate an instance of this interactive protocol and, depending on the modeled security property, may drop, modify, or inject messages in the communication.

Security properties are defined through games, which are experimental frameworks challenging the adversary to breach desired security properties. These games consist of multiple phases, beginning with a setup phase where the challenger generates parameters (such as encryption keys) for the game. In the subsequent query/response phase, the adversary receives parameters (typically including public keys but excluding secrets) from the setup phase and operates with access to oracles. The challenger answers the adversary's oracle queries during this phase.

Once the adversary's program terminates, the challenger invokes it with a challenge, requiring the adversary to compute a final response. Depending on this response, the challenger decides whether the adversary wins the game, returning 0 if the adversary loses and 1 if the adversary wins.

A game for the existential unforgeability of signatures could be formulated like this:

$\mathbf{Exp}_A^{EUF}(1^\lambda)$:

1. $(sk, pk) \leftarrow \text{KeyGen}(1^\lambda)$
2. $(\sigma, m) \leftarrow A^{\text{OSign}(\cdot)}(pk)$
   (Run the adversary with input pk and access to the OSign oracle.)
3. If the adversary has called OSign$(\cdot)$ with m as argument, return 0.
4. Return Verify(pk, σ, m).

Here, the adversary is run once, with access to the signing oracle. Depending on which definition of OSign is chosen, the game models existential unforgeability under chosen message attack (EUF-CMA) or existential unforgeability under known message attack (EUF-KMA).

The following modification to the game would model selective unforgeability (SUF-CMA / SUF-KMA):

$\mathbf{Exp}_A^{SUF}(1^\lambda)$:

1. $m \leftarrow A()$
2. $(sk, pk) \leftarrow \text{KeyGen}(1^\lambda)$
3. $\sigma \leftarrow A^{\text{OSign}(\cdot)}(pk, m)$
4. If the adversary has called OSign$(\cdot)$ with m as argument, return 0.
5. Return Verify(pk, σ, m).

Here, the adversary has to choose a message to forge a signature for before knowing the message verification key.

After having defined the game, we can now define a security property based on the probability of the adversary winning the game: we say that a signature scheme is secure against existential unforgeability attacks if for every adversary A (i.e., a polynomial-time probabilistic Turing machine program), the success probability

$$\Pr\left[\text{Exp}_A^{\text{EUF}}(1^\lambda) = 1\right]$$

of A in the EUF game is negligible (i.e., grows less fast with λ than the inverse of any polynomial in λ). Note that the EUF and SUF games are related in the following way: an adversary against the SUF game can be easily transformed into an adversary against the EUF game, while the converse does not necessarily hold. Often, security properties are defined in terms of the advantage of the adversary. The advantage is a measure of how likely the adversary is to win against the real cryptographic protocol, compared to a perfectly secure version of the protocol. For example, let $\mathrm{Exp}_A^{\mathrm{BIT}}()$ be a game where the adversary has to guess the next bit in the output of a pseudo-random number generator (PRNG). The idealized functionality would be a real random number generator, where the adversary's chance of a correct guess is $\frac{1}{2}$. Thus, the adversary's advantage is

$$\Pr\left[\mathrm{Exp}_A^{\mathrm{BIT}}()\right] - \frac{1}{2}.$$

Note that the definition of advantage depends on the game. The above definition, for example, would not work if the adversary had a way to "voluntarily" lose the game by querying an oracle in a forbidden way.

6.1.2 Assumptions, Reductions and Game Hopping

The goal of a security proof is to transform an attacker against the protocol under consideration into an attacker against the security of an underlying assumption. Typical examples for common assumptions might be:

- the difficulty of the decisional/computational Diffie–Hellman problem (nicely described by [Bon98])
- existential unforgeability under chosen message attack (EUF-CMA) of a signature scheme [GMR88]
- indistinguishability against chosen-plaintext attacks (IND-CPA) of a symmetric encryption algorithm [Bel+98]

To construct a reduction from an adversary A against P to an adversary against Q, it is necessary to specify a program R that both interacts as an adversary with the challenger for Q, but at the same time acts as a challenger for the adversary against P. Most importantly, R can choose how to respond to oracle queries from the adversary, as long as R faithfully simulates a challenger for P. The reduction must be efficient, i.e., R must still be a polynomial-time algorithm. A well-known example for a non-trivial reduction proof is the security proof of FDH-RSA signatures [Cor00].

In practice, reduction proofs are often complex and hard to verify. Game hopping has become a popular technique to manage the complexity of security proofs. The idea behind game hopping proofs is to make a sequence of small modifications starting from the initial game, until you arrive at a game where the success probability for the adversary becomes obvious, for example, because the winning state for the adversary becomes unreachable in the code that defines the final game, or because all values the adversary can observe to make a decision are drawn from a truly random and uniform distribution. Each hop modifies the game in a way such that the success probability of game G_n and game G_{n+1} is negligibly close.

Useful techniques for hops are, for example:

- Bridging hops, where the game is syntactically changed but remains semantically equivalent, i.e., $\Pr[G_n = 1] = \Pr[G_n = 1]$.
- Indistinguishability hops, where some distribution is changed in a way that an adversary that could distinguish between two adjacent games could be turned into an adversary that distinguishes the two distributions. If the success probability to distinguish between those two distributions is ϵ, then $|\Pr[G_n = 1] - \Pr[G_n = 1]| \leq \epsilon$.
- Hops based on small failure events. Here adjacent games proceed identically, until in one of the games a detectable failure event F (such as an adversary visibly forging a signature) occurs. Both games must proceed the same if F does not occur. Then it is easy to show [Shoo4] that

$$|\Pr[G_n = 1] - \Pr[G_n = 1]| \leq \Pr[F].$$

A tutorial introduction to game hopping is given by Shoup [Shoo4], while a more formal treatment with a focus on "games as code" can be found in [BR06]. A version of the FDH-RSA security proof based on game hopping was generated with an automated theorem prover by Blanchet and Pointcheval [BP06].

6.1.3 Notation

We prefix public and secret keys with *pk* and *sk*, and write $x \leftarrow S$ to randomly select an element x from the set S with uniform probability.

6.2 Model and Syntax for our System

We consider a payment system with a single, static exchange and multiple, dynamically created customers and merchants. The subset of the full our protocol that we model includes withdrawing digital coins, spending them with merchants, and subsequently depositing them at the exchange, as well as obtaining unlinkable change for partially spent coins with an online "refresh" protocol.

The exchange offers digital coins in multiple denominations, and every denomination has an associated financial value; this mapping is not chosen by the adversary but is a system parameter. We mostly ignore the denomination values here, including their impact on anonymity, in keeping with existing literature [CLM07], [PST17]. For anonymity, we believe this amounts to assuming that all customers have similar financial behavior. We note logarithmic storage, computation, and bandwidth demands for denominations distributed by powers of a fixed constant, like two.

We do not model fees taken by the exchange. Reserves are also omitted. Instead of maintaining a reserve balance, withdrawals of different denominations are tracked, effectively assuming every customer has unlimited funds.

Coins can be partially spent by specifying a fraction $0 < f \leq 1$ of the total value associated with the coin's denomination. Unlinkable change below the smallest denomination cannot be given. In practice, the unspendable, residual value should be seen as an additional fee charged by the exchange.

Spending multiple coins is modeled non-atomically: to spend (fractions of) multiple coins, they must be spent one-by-one. The individual spend/deposit operations are correlated by a unique identifier for the transaction. In practice, this identifier is the hash $transactionId = H(contractTerms)$ of the contract terms. Contract terms include a nonce to make them unique, that merchant and customer agreed upon. Note that this transaction identifier and the correlation between multiple spend operations for one payment need not be disclosed to the exchange (it might, however, be necessary to reveal during a detailed tax audit of the merchant): When spending the i-th coin for the transaction with the identifier $transactionId$, messages to the exchange would only contain $H(i \| transactionId)$. This is preferable for merchants that might not want to disclose to the exchange the individual prices of products they sell to customers, but only the total transaction volume over time. For simplicity, we do not include this extra feature in our model.

Our system model tracks the total amount (financial value) of coins withdrawn by each customer. Customers are identified by their public key $pk_{Customer}$. Every customer's wallet keeps track of the following data:

- `wallet[pkCustomer]` contains sets of the customer's coin records, which individually consist of the coin key pair, denomination, and exchange's signature.
- `acceptedContracts[pkCustomer]` contains the sets of transaction identifiers accepted by the customer during spending operations, together with coins spent for it and their contributions $0 < f \leq 1$.
- `withdrawIds[pkCustomer]` contains the withdraw identifiers of all withdraw oper-

ations that were created for this customer.

- `refreshIds[pkCustomer]` contains the refresh identifiers of all refresh operations that were created for this customer.

The exchange in our model keeps track of the following data:

- `withdrawn[pkCustomer]` contains the total amount withdrawn by each customer, i.e., the sum of the financial value of the denominations for all coins that were withdrawn by `pkCustomer`.
- The overspending database of the exchange is modeled by `deposited[pkCoin]` and `refreshed[pkCoin]`, which record deposit and refresh operations respectively on each coin. Note that since partial deposits and multiple refreshes to smaller denominations are possible, one deposit and multiple refresh operations can be recorded for a single coin.

We say that a coin is fresh if it appears in neither the `deposited` nor `refreshed` lists nor in `acceptedContracts`. We say that a coin is being overspent if recording an operation in `deposited` or `refreshed` would cause the total spent value from both lists to exceed the value of the coin's denomination. Note that the adversary does not have direct read or write access to these values; instead, the adversary needs to use the oracles (defined later) to interact with the system.

We parameterize our system with two security parameters: The general security parameter λ, and the refresh security parameter κ. While λ determines the length of keys and thus the security level, using a larger κ will only decrease the success chance of malicious merchants conspiring with customers to obtain unreported (and thus untaxable) income.

6.2.1 Algorithms

The Our e-cash scheme is modeled by the following probabilistic[1] algorithms and interactive protocols. The notation $P(X_1, \dots, X_n)$ stands for a party $P \in \{E, C, M\}$ (Exchange, Customer, and Merchant respectively) in an interactive protocol, with $X_1, \dots, X_n$ being the (possibly private) inputs contributed by the party to the protocol. Interactive protocols can access the state maintained by party P.

While the adversary can freely execute the interactive protocols by creating their own parties, the adversary is not given direct access to the private data of parties maintained by the challenger in the security games we define later.

[1]Polynomial-time algorithms and interactive protocols.

- `ExchangeKeygen`(1^λ, 1^κ, D) $\rightarrow$ (sks_E, pks_E): Algorithm executed to generate keys for the exchange, with general security parameter λ and refresh security parameter κ, both given as unary numbers. The denomination specification $D = d_1, \ldots, d_n$ is a finite sequence of positive rational numbers that defines the financial value of each generated denomination key pair. We henceforth use D to refer to some appropriate denomination specification, but our analysis is independent of a particular choice of D.

  The algorithm generates the exchange's master signing key pair (sk_{ESig}, pk_{ESig}) and denomination secret and public keys $(sk_{D_1}, \ldots, sk_{D_n}), (pk_{D_1}, \ldots, pk_{D_n})$. We write $D(pk_{D_i})$, where $D : \{pk_{D_i}\} \rightarrow D$ to look up the financial value of denomination pk_{D_i}.

  We collectively refer to the exchange's secrets by $\text{sk}_s E$ and to the exchange's public keys together with D by $\text{pk}_s E$.

- `CustomerKeygen`(1^λ, 1^κ) $\rightarrow$ ($\text{sk}_{Customer}$, $\text{pk}_{Customer}$): Key generation algorithm for customers with security parameters λ and κ.

- `MerchantKeygen`(1^λ, 1^κ) $\rightarrow$ ($\text{sk}_{Merchant}$, $\text{pk}_{Merchant}$): Key generation algorithm for merchants. Typically the same as `CustomerKeygen`.

- `WithdrawRequest`(E(sks_E, $\text{pk}_{Customer}$), C($\text{sk}_{Customer}$, pks_E, pk_D)) $\rightarrow$ (TWR, wid): Interactive protocol between the exchange and a customer that initiates withdrawing a single coin of a particular denomination. The customer obtains a withdraw identifier wid from the protocol execution and stores it in withdrawIds[pkCustomer].

  The `WithdrawRequest` protocol only initiates a withdrawal. The coin is only obtained and stored in the customer's wallet by executing the `WithdrawPickup` protocol on the withdraw identifier wid. The customer and exchange persistently store additional state (if required by the instantiation) such that the customer can use `WithdrawPickup` to complete withdrawal or to complete a previously interrupted or unfinished withdrawal. Returns a protocol transcript TWR of all messages exchanged between the exchange and customer, as well as the withdraw identifier wid.

- `WithdrawPickup`(E(sks_E, $\text{pk}_{Customer}$), C($\text{sk}_{Customer}$, pks_E, wid)) $\rightarrow$ (TWP, coin): Interactive protocol between the exchange and a customer to obtain the coin from a withdraw operation started with `WithdrawRequest`, identified by the withdraw identifier wid. The first time `WithdrawPickup` is run with a particular withdraw identifier wid, the exchange increments withdrawn[pkCustomer] by D(pk_D), where pk$_D$ is the denomination requested in the corresponding `WithdrawRequest` execution. How exactly pk$_D$ is restored depends on the particular instantiation.

The resulting coin

$$\text{coin} = (sk_{Coin}, pk_{Coin}, pk_D, \text{coinCert}),$$

consisting of secret key sk$_{Coin}$, public key pk$_{Coin}$, denomination public key pk$_D$ and certificate coinCert from the exchange, is stored in the customer's wallet wallet[pk$_{Customer}$]. Executing the `WithdrawPickup` protocol multiple times with the same customer and the same withdraw identifier does not result in any change of the customer's withdraw balance withdrawn[pkCustomer], and results in (re-)adding the same coin to the customer's wallet. Returns a protocol transcript TWP of all messages exchanged between the exchange and customer.

- `Spend`(transactionId, f, coin, pk$_{Merchant}$) $\rightarrow$ depositPermission: Algorithm to produce and sign a deposit permission depositPermission for a coin under a particular transaction identifier. The fraction $0 < f \leq 1$ determines the fraction of the coin's initial value that will be spent. The contents of the deposit permission depend on the instantiation, but it must be possible to derive the public coin identifier pk$_{Coin}$ from them.

- `Deposit`(E(sks$_E$, pk$_{Merchant}$), M(sk$_{Merchant}$, pks$_E$, depositPermission)) $\rightarrow$ TD: Interactive protocol between the exchange and a merchant. From the deposit permission, we obtain the pk$_{Coin}$ of the coin to be deposited. If pk$_{Coin}$ is being overspent, the protocol is aborted with an error message to the merchant. On success, we add depositPermission to deposited[pk$_{Coin}$]. Returns a protocol transcript TD of all messages exchanged between the exchange and merchant.

- `RefreshRequest`(E(sks$_E$), C(pk$_{Customer}$, pks$_E$, coin$_0$, pk$_{D_u}$)) $\rightarrow$ (T$_{RR}$, rid): Interactive protocol between the exchange and customer that initiates a refresh of coin$_0$. Together with `RefreshPickup`, it allows the customer to convert D(pk$_{D_u}$) of the remaining value on coin

$$\text{coin}_0 = (sk_{Coin_0}, pk_{Coin_0}, pk_{D_0}, \text{coinCert}_0)$$

into a new, unlinkable coin coinu of denomination pk$_{D_u}$. Multiple refreshes on the same coin are allowed, but each run subtracts the respective financial value of coinu from the remaining value of coin$_0$. The customer only records the refresh operation identifier rid in refreshIds[pkCustomer], but does not yet obtain the new coin. To obtain the new coin, `RefreshPickup` must be used. Returns the protocol transcript T$_{RR}$ and a refresh identifier rid.

- `RefreshPickup`(E(sks$_E$, pk$_{Customer}$), C(sk$_{Customer}$, pks$_E$, rid)) $\rightarrow$ (T$_{RP}$, coinu): Interactive protocol between the exchange and customer to obtain the new coin for a refresh operation previously started with `RefreshRequest`, identified by the refresh identifier rid. The exchange learns the target denomination pk$_{D_u}$ and signed source coin (pk$_{Coin_0}$, pk$_{D_0}$, coinCert$_0$). If the source coin is invalid, the exchange aborts the protocol. The first time `RefreshPickup` is run for a particular refresh identifier, the exchange records a refresh operation of value D(pk$_{D_u}$) in refreshed[pk$_{Coin_0}$]. If pk$_{Coin_0}$ is being overspent, the refresh operation is not recorded in refreshed[pk$_{Coin_0}$], the exchange sends the customer the protocol transcript of the previous deposits and refreshes and aborts the protocol. If the customer C plays honestly in `RefreshRequest` and `RefreshPickup`, the unlinkable coin coinu they obtain as change will be stored in their wallet wallet[pk$_{Customer}$]. If C is caught playing dishonestly, the `RefreshPickup` protocol aborts. An honest customer must be able to repeat a `RefreshPickup` with the same rid multiple times and (re-)obtain the same coin, even if previous `RefreshPickup` executions were aborted. Returns a protocol transcript T$_{RP}$.

- `Link`(E(sks$_E$), C(sk$_{Customer}$, pks$_E$, coin$_0$)) $\rightarrow$ (T, (coin$_1$, $\ldots$, coin$_n$)): Interactive protocol between the exchange and customer. If coin$_0$ is a coin that was refreshed, the customer can recompute all the coins obtained from previous refreshes on coin$_0$, with data obtained from the exchange during the protocol. These coins are added to the customer's wallet wallet[pk$_{Customer}$] and returned together with the protocol transcript.

6.2.2 Oracles

We now specify how the adversary can interact with the system by defining oracles. Oracles are queried by the adversary, and upon a query, the challenger will act according to the oracle's specification. Note that the adversary for the different security games is run with specific oracles and does not necessarily have access to all oracles simultaneously.

We refer to customers in the parameters to an oracle query simply by their public key. The adversary needs the ability to refer to coins to trigger operations such as spending and refresh, but to model anonymity we cannot give the adversary access to the coins' public keys directly. Therefore, we allow the adversary to use the (successful) transcripts of the withdraw, refresh, and link protocols to indirectly refer to coins. We refer to this as a coin handle H. Since the execution of a link protocol results in a transcript T that can contain multiple coins, the adversary needs to select a particular coin from the transcript via the index i as

$$H = (T, i).$$

The respective oracle tries to find the coin that resulted from the transcript given by the adversary. If the transcript has not been seen before in the execution of a link, refresh, or withdraw protocol, or the index for a link transcript is invalid, the oracle returns an error to the adversary.

In oracles that trigger the execution of one of the interactive protocols defined in Section 3.2.1, we give the adversary the ability to actively control the communication channels between the exchange, customers, and merchants; i.e., the adversary can effectively record, drop, modify, and inject messages during the execution of the interactive protocol. Note that this allows the adversary to leave the execution of an interactive protocol in an unfinished state, where one or more parties are still waiting for messages. We use I to refer to a handle to interactive protocols where the adversary can send and receive messages.

- $O\text{AddCustomer}() \rightarrow pkCustomer$: Generates a key pair $(skCustomer, pkCustomer)$ using the CustomerKeygen algorithm, and sets

  $$\begin{aligned} \text{withdrawn}[pkCustomer] &:= 0 \\ \text{acceptedContracts}[pkCustomer] &:= \{\} \\ \text{wallet}[pkCustomer] &:= \{\} \\ \text{withdrawIds}[pkCustomer] &:= \{\} \\ \text{refreshIds}[pkCustomer] &:= \{\}. \end{aligned}$$

  Returns the public key of the newly created customer.

- $O\text{AddMerchant}() \rightarrow pkMerchant$: Generates a key pair $(skMerchant, pkMerchant)$ using the MerchantKeygen algorithm. Returns the public key of the newly created merchant.

- $O\text{SendMessage}(I, P_1, P_2, m) \rightarrow ()$: Sends message m on the channel from party P_1 to party P_2 in the execution of interactive protocol I. The oracle does not have a return value.

- $O\text{ReceiveMessage}(I, P_1, P_2) \rightarrow m$: Reads message m in the channel from party P_1 to party P_2 in the execution of interactive protocol I. If no message is queued in the channel, return $m = \perp$.

- $O\text{WithdrawRequest}(pkCustomer, pkD) \rightarrow I$: Triggers the execution of the WithdrawRequest protocol, giving the adversary full control of the communication channels between the customer and exchange.

- OWithdrawPickup$(pkCustomer, pkD, T) \to I$: Triggers the execution of the WithdrawPickup protocol, additionally giving the adversary full control of the communication channels between the customer and exchange. The customer and withdraw identifier wid are obtained from the WithdrawRequest transcript T.

- ORefreshRequest$(H, pkD) \to I$: Triggers the execution of the RefreshRequest protocol with the coin identified by coin handle H, additionally giving the adversary full control over the communication channels between the customer and exchange.

- ORefreshPickup$(T) \to I$: Triggers the execution of the RefreshPickup protocol, where the customer and refresh identifier rid are obtained from the RefreshRequest protocol transcript T. Additionally gives the adversary full control over the communication channels between the customer and exchange.

- OLink$(H) \to I$: Triggers the execution of the Link protocol for the coin referenced by handle H, additionally giving the adversary full control over the communication channels between the customer and exchange.

- OSpend$(transactionId, pkCustomer, H, pkMerchant) \to depositPermission$: Makes a customer sign a deposit permission over a coin identified by handle H. Returns the deposit permission on success, or $\perp$ if H is not a coin handle that identifies a coin. Note that OSpend can be used to generate deposit permissions that, when deposited, would result in an error due to overspending. It adds $(transactionId, depositPermission)$ to acceptedContracts$[pkCustomer]$.

- OShare$(H, pkCustomer) \to ()$: Shares a coin (identified by handle H) with the customer identified by $pkCustomer$, i.e., puts the coin identified by H into wallet$[pkCustomer]$. Intended to be used by the adversary in attempts to violate income transparency. Does not have a return value. Note that this trivially violates anonymity (by sharing with a corrupted customer), thus the usage must be restricted in some games.

- $$\begin{aligned} O\text{CorruptCustomer}(pkCustomer) \to \\ &(skCustomer, wallet[pkCustomer], \\ &\quad acceptedContracts[pkCustomer], \\ &\quad refreshIds[pkCustomer], \\ &\quad withdrawIds[pkCustomer]) : \end{aligned}$$

Used by the adversary to corrupt a customer, giving the adversary access to the customer's secret key, wallet, withdraw/refresh identifiers, and accepted contracts. Permanently marks the customer as corrupted. There is nothing "special" about corrupted customers, other than that the adversary has used OCorruptCustomer on them in the past. The adversary cannot modify corrupted customer's wallets directly, and must use the oracle again to obtain an updated view on the corrupted customer's private data.

- $O\text{Deposit}(\mathit{depositPermission}) \to I$: Triggers the execution of the Deposit protocol, additionally giving the adversary full control over the communication channels between the merchant and exchange. Returns an error if the deposit permission is addressed to a merchant that was not registered with OAddMerchant. This oracle does not give the adversary new information but is used to model the situation where there might be multiple conflicting deposit permissions generated via OSpend, but only a limited number can be deposited.

We write OSolution for the set of all the oracles we just defined, and $O\text{NoShare} := O\text{Solution} - O\text{Share}$ for all oracles except the share oracle.

The exchange does not need to be corrupted with an oracle. A corrupted exchange is modeled by giving the adversary the appropriate oracles and the exchange secret key from the exchange key generation. If the adversary determines the exchange's secret key during the setup, invoking OWithdrawRequest, OWithdrawPickup, ORefreshRequest, ORefreshPickup, or OLink can be seen as the adversary playing the exchange. Since the adversary is an active man-in-the-middle in these oracles, it can drop messages to the simulated exchange and make up its own response. If the adversary calls these oracles with a corrupted customer, the adversary plays as the customer.

6.3 Games

We now define four security games (anonymity, conservation, unforgeability, and income transparency) that are later used to define the security properties for Our solution. Similar to [BR06], we assume that the game and adversary terminate in finite time, and thus random choices made by the challenger and adversary can be taken from a finite sample space.

All games except income transparency return 1 to indicate that the adversary has won and 0 to indicate that the adversary has lost. The income transparency game returns 0 if the adversary has lost, and a positive "laundering ratio" if the adversary won.

6.3.1 Anonymity

Intuitively, an adversary A (controlling the exchange and merchants) wins the anonymity game if they have a non-negligible advantage in correlating spending operations with the withdrawal or refresh operations that created a coin used in the spending operation.

Let b be the bit that will determine the mapping between customers and spend operations, which the adversary must guess. We define a helper procedure

$$\text{Refresh}(E(sks_E), C(pk_{Customer}, pks_E, coin_0)) \to R$$

that refreshes the whole remaining amount on $coin_0$ with repeated application of RefreshRequest and RefreshPickup using the smallest possible set of target denominations, and returns all protocol transcripts in R.

```
$Exp_{\mathcal{A}}^{anon}(1^\lambda, 1^\kappa, b)$:
  1. $(\mathsf{sksE}, \mathsf{pksE}, \mathsf{skM}, \mathsf{pkM}) \leftarrow \mathcal{A}()$
  2. $(\mathsf{pkCustomer}_0, \mathsf{pkCustomer}_1, \mathsf{transactionId}_0, \mathsf{transactionId}_1, f) \leftarrow \mathcal{A}^{\mathcal{O}\textsc{NoShare}}()$
  3. Select distinct fresh coins
         $\mathsf{coin}_0 \in \mathsf{wallet}[\mathsf{pkCustomer}_0]$
         $\mathsf{coin}_1 \in \mathsf{wallet}[\mathsf{pkCustomer}_1]$
     Return 0 if either $\mathsf{pkCustomer}_0$ or $\mathsf{pkCustomer}_1$ are not registered customers with sufficient fresh coins.
  4. For $i \in \{0,1\}$ run
         $\mathsf{dp_i} \leftarrow \mathsf{Spend}(\mathsf{transactionId}_i, f, \mathsf{coin}_{i-b}, \mathsf{pkM})$
         $\mathsf{Deposit}(\mathcal{A}(), \mathcal{M}(\mathsf{skM}, \mathsf{pksE}, \mathsf{dp}_i))$
         $\mathfrak{R}_i \leftarrow \mathsf{Refresh}(\mathcal{A}(), \mathcal{C}(\mathsf{pkCustomer}_i, \mathsf{pksE}, \mathsf{coin}_{i-b}))$
  5. $b' \leftarrow \mathcal{A}^{\mathcal{O}\textsc{NoShare}}(\mathfrak{R}_0, \mathfrak{R}_1)$
  6. Return 0 if $\mathcal{O}\mathsf{Spend}$ was used by the adversary on the coin handles for $\mathsf{coin}_0$ or $\mathsf{coin}_1$ or $\mathcal{O}\mathsf{CorruptCustomer}$ was used on $\mathsf{pkCustomer}_0$ or $\mathsf{pkCustomer}_1$.
  7. If $b = b'$ return 1, otherwise return 0.
```

Note that unlike some other anonymity games defined in the literature (such as [PST17]), our anonymity game always lets both customers spend in order to avoid having to hide the missing coin in one customer's wallet from the adversary.

6.3.2 Conservation

The adversary wins the conservation game if it can bring an honest customer in a situation where the spendable financial value left in the user's wallet plus the value spent for transactions known to the customer is less than the value withdrawn by the same customer through the exchange. In practice, this property is necessary to guarantee that aborted or partially completed withdrawals, payments or refreshes, as well as other (transient) misbehavior from the exchange or merchant do not result in the customer losing money.

$\mathbf{Exp}_{A}^{\text{conserv}}(1^\lambda, 1^\kappa)$:
1.$(sks_E, pks_E) \leftarrow \text{ExchangeKeygen}(1^\lambda, 1^\kappa, M)$
2. $pk_{Customer} \leftarrow A^{ONoShare}(pks_E)$
3. Return 0 if $pk_{Customer}$ is a corrupted user.
4. Run WithdrawPickup for each withdraw identifier *wid* and RefreshPickup for each refresh identifier *rid* that the user has recorded in *withdrawIds* and *refreshIds*. Run Deposit for all deposit permissions in *acceptedContracts*.
5. Let v_C be the total financial value left on valid coins in $wallet[pk_{Customer}]$, i.e., the denominated values minus the spend/refresh operations recorded in the exchange's database. Let v_S be the total financial value of contracts in $acceptedContracts[pk_{Customer}]$.
6. Return 1 if $withdrawn[pk_{Customer}] > v_C + v_S$.

Hence we ensure that:

- if a coin was spent, it was spent for a contract that the customer knows about, i.e., in practice the customer could prove that they "own" what they paid for,
- if a coin was refreshed, the customer "owns" the resulting coins, even if the operation was aborted, and
- if the customer withdraws, they can always obtain a coin whenever the exchange accounted for a withdrawal, even when protocol executions are intermittently aborted.

Note that we do not give the adversary access to the *OShare* oracle, since that would trivially allow the adversary to win the conservation game. In practice, conservation only holds for customers that do not share coins with parties that they do not fully trust.

6.3.3 Unforgeability

Intuitively, adversarial customers win if they can obtain more valid coins than they legitimately withdraw.

$\mathbf{Exp}_A^{\text{forge}}(1^\lambda, 1^\kappa)$:
1.$(sk_E, pk_E) \leftarrow$ ExchangeKeygen()
2. $(C_0, \ldots, C_\ell) \leftarrow A^{OAll}(pk_{Exchange})$
3. Return 0 if any C_i is not of the form $(sk_{Coin}, pk_{Coin}, pk_D, coinCert)$ or any $coinCert$ is not a valid signature by pk_D on the respective pk_{Coin}.
4. Return 1 if the sum of the unspent value of valid coins in $C_0, \ldots, C_\ell$ exceeds the amount withdrawn by corrupted customers; return 0 otherwise.

6.3.4 Income Transparency

Intuitively, the adversary wins if coins are in exclusive control of corrupted customers, but the exchange has no record of withdrawal or spending for them. This presumes that the adversary cannot delete from non-corrupted customers' wallets, even though it can use oracles to force protocol interactions of non-corrupted customers.

For practical e-cash systems, income transparency disincentivizes the emergence of "black markets" among mutually distrusting customers, where currency circulates without the transactions being visible. This is in contrast to some other proposed e-cash systems and cryptocurrencies, where disintermediation is an explicit goal. The Link protocol introduces the threat of losing exclusive control of coins (despite having the option to refresh them) that were received without being visible as income to the exchange.

$\mathbf{Exp}_A^{\text{income}}(1^\lambda, 1^\kappa)$:
1.$(sk_E, pk_E) \leftarrow$ ExchangeKeygen()
2. $(coin_1, \ldots, coin_\ell) \leftarrow A^{OAll}(pk_{Exchange})$
(The $coin_i$ must be coins, including secret key and signature by the denomination, for the adversary to win. However, these coins need not be present in any honest or corrupted customer's wallet.)
3. Augment the wallets of all non-corrupted customers with their transitive closure using the Link protocol. Mark all remaining value on coins in wallets of non-corrupted customers as spent in the exchange's database.
4. Let L denote the sum of unspent value on valid coins in $(coin_1, \ldots, coin_\ell)$, after accounting for the previous update of the exchange's database. Also, let w' be the sum of coins withdrawn by corrupted customers. Then $p := L - w'$ gives the adversary's untaxed income.

5. Let w be the sum of coins withdrawn by non-corrupted customers, and s be the value marked as spent by non-corrupted customers, so that $b := w - s$ gives the coins lost during refresh, that is, the losses incurred attempting to hide income.
6. If $b + p \neq 0$, return $\frac{p}{b+p}$, i.e., the laundering ratio for attempting to obtain untaxed income. Otherwise, return 0.

6.4 Security Definitions

We now give security definitions based upon the games defined in the previous section. Recall that λ is the general security parameter, and κ is the security parameter for income transparency. A polynomial-time adversary is implied to be polynomial in $\lambda + \kappa$.

Definition 1 (Anonymity). *We say that an e-cash scheme satisfies anonymity if the success probability*

$$\Pr\left[b \leftarrow \{0,1\} : Exp_{A}^{anon}(1^{\lambda}, 1^{\kappa}, b) = 1\right]$$

of the anonymity game is negligibly close to $\frac{1}{2}$ for any polynomial-time adversary A.

Definition 2 (Conservation). *We say that an e-cash scheme satisfies conservation if the success probability*

$$\Pr\left[Exp_{A}^{conserv}(1^{\lambda}, 1^{\kappa}) = 1\right]$$

of the conservation game is negligible for any polynomial-time adversary A.

Definition 3 (Unforgeability). *We say that an e-cash scheme satisfies unforgeability if the success probability*

$$\Pr\left[Exp_{A}^{forge}(1^{\lambda}, 1^{\kappa}) = 1\right]$$

of the unforgeability game is negligible for any polynomial-time adversary A.

Definition 4 (Strong Income Transparency). *We say that an e-cash scheme satisfies strong income transparency if the success probability*

$$\Pr\left[Exp_{A}^{income}(1^{\lambda}, 1^{\kappa}) \neq 0\right]$$

for the income transparency game is negligible for any polynomial-time adversary A. The adversary is said to win one execution of the strong income transparency game if the game's return value is non-zero, i.e., there was at least one successful attempt to obtain untaxed income.

Definition 5 (Weak Income Transparency). *We say that an e-cash scheme satisfies weak income transparency if, for any polynomial-time adversary A, the return value of the* *income transparency game satisfies*

$$\mathbb{E}\left[Exp_A^{income}(1^\lambda, 1^\kappa)\right] \leq \frac{1}{\kappa}. \tag{3.1}$$

In (3.1), the expectation runs over any probability space used by the adversary and challenger. For some instantiations, e.g., ones based on zero-knowledge proofs, κ might be a security parameter in the traditional sense. However, for an e-cash scheme to be useful in practice, the adversary does not need to have only negligible success probability to win the income transparency game. It suffices that the financial losses of the adversary in the game are a deterrent, after all, our purpose of the game is to characterize tax evasion.

Our solution does not fulfill strong income transparency, since for Our solution κ must be a small cut-and-choose parameter, as the complexity of our cut-and-choose protocol grows linearly with κ. Instead, we show that Our solution satisfies weak income transparency, which is a statement about the adversary's financial loss when winning the game instead of the winning probability. The return-on-investment (represented by the game's return value) is bounded by $\frac{1}{\kappa}$. We still characterize strong income transparency, since it might be useful for other instantiations that provide more absolute guarantees.

6.5 Instantiation

We now give a concrete instantiation that is used in the implementation of Our solution for the schemes `BlindSign`, `CoinSignKx`, and `Sign`.

For `BlindSign`, we use RSA-FDH blind signatures [Cha83] [BR96]. From the information-theoretic security of blinding, the computational blindness property follows directly. For the unforgeability property, we additionally rely on the RSA-KTI assumption as discussed in [Bel+03]. Note that since the blinding step in RSA blind signatures is non-interactive, storage and verification of the transcript is omitted in refresh and link.

We instantiate `CoinSignKx` with signatures and key exchange operations on elliptic curves in Edwards form, where the same key is used for signatures and the Diffie–Hellman key exchange operations. In practice, we use Ed25519 [Ber+12] / Curve25519 [Ber06] for $\lambda = 256$. We caution that some other elliptic curve key exchange implementations might not satisfy the completeness property that we require, due to the lack of complete addition laws.

For `Sign`, we use elliptic-curve signatures, concretely Ed25519. For the collision-resistant hash function H, we use SHA-512 [H306] and HKDF [KE10] as a PRF.

Bibliography

[Ady16] Adyen. The Global E-Commerce Payments Guide. 2016 (cit. on p. 2).

[And+18] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, et al. "Hyperledger fabric: a distributed operating system for permissioned blockchains". In: Proceedings of the Thirteenth EuroSys Conference. ACM. 2018, p. 30 (cit. on p. 30).

[Ano99] Anonymous. How DigiCash Blew Everything. 1999 (cit. on p. 27).

[AO00] Masayuki Abe and Tatsuaki Okamoto. "Provably secure partially blind signatures". In: Annual International Cryptology Conference. Springer. 2000, pp. 271–286 (cit. on pp. 27, 48).

[Arn+18] Douglas W Arner, Dirk A Zetzsche, Ross P Buckley, and Janos Nathan Barberis. "The Identity Challenge in Finance: From Analogue Identity to Digitized Identification to Digital KYC Utilities". In: European Banking Institute (2018) (cit. on p. 64).

[ASM11] Man Ho Au, Willy Susilo, and Yi Mu. "Electronic cash with anonymous user suspension". In: Australasian Conference on Information Security and Privacy. Springer. 2011, pp. 172–188 (cit. on pp. 4, 27).

[Bad15] Heinz-Peter Bader. France steps up monitoring of cash payments to fight low-cost terrorism. *http://www.reuters.com/article/2015/03/18/us-france-securityfinancing-idUSKBN0ME14720150318*. Mar. 2015 (cit. on p. 14).

[Bar11] Adam Barth. The Web Origin Concept. RFC 6454. Dec. 2011. url: *https://rfceditor.org/rfc/rfc6454.txt* (cit. on p. 68).

[Bel+03] Bellare, Namprempre, Pointcheval, and Semanko. "The One-More-RSA-Inversion Problems and the Security of Chaum's Blind Signature

Scheme". In: Journal of Cryptology 16.3 (June 2003), pp. 185–215. url: *https://doi.org/10.1007/s00145-002-0120-1* (cit. on pp. 52, 62).

[Bel+98] Mihir Bellare, Anand Desai, David Pointcheval, and Phillip Rogaway. "Relations among notions of security for public-key encryption schemes". In: Annual International Cryptology Conference. Springer. 1998, pp. 26–45 (cit. on p. 38).

[Ben+14] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. "Zerocash: Decentralized Anonymous Payments from Bitcoin". In: IEEE Symposium on Security Privacy. 2014 (cit. on p. 30).

[Ben+18] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. "Scalable, transparent, and post-quantum secure computational integrity". In: Cryptol. ePrint Arch., Tech. Rep 46 (2018), p. 2018 (cit. on p. 30).

[Ber+12] Daniel J Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. "High-speed high-security signatures". In: Journal of Cryptographic Engineering 2.2 (2012), pp. 77–89 (cit. on p. 52).

[Ber06] Daniel J Bernstein. "Curve25519: new Diffie-Hellman speed records". In: International Workshop on Public Key Cryptography. Springer. 2006, pp. 207–228 (cit. on pp. 52, 62).

[CGH06] Sébastien Canard, Aline Gouget, and Emeline Hufschmitt. "A handy multi-coupon system". In: International Conference on Applied Cryptography and Network Security. Springer. 2006, pp. 66–81 (cit. on p. 25).

[Cha+89] David Chaum, Bert den Boer, Eugène van Heyst, Stig Mjølsnes, and Adri Steenbeek. "Efficient offline electronic checks". In: Workshop on the theory and application of of cryptographic techniques. Springer. 1989, pp. 294–301 (cit. on p. 25).

[Cha83] David Chaum. "Blind Signatures for Untraceable Payments." In: *Advances in Cryptology: Proceedings of Crypto '82*. Ed. by David Chaum, Ronald L. Rivest, and Alan T. Sherman. Boston, MA: Springer US, 1983, pp. 199–203. DOI: `https://doi.org/10.1007/978-1-4757-0602-4_18` (cit. on pp. 3, 6, 13, 25, 28, 52).

[Che+14] Yuchung Cheng, Jerry Chu, Sivasankar Radhakrishnan, and Arvind Jain. TCP Fast Open. RFC 7413. Dec. 2014. url: *https://rfc-editor.org/rfc/rfc7413.txt* (cit. on p. 101).

[CHL05] Jan Camenisch, Susan Hohenberger, and Anna Lysyanskaya. "Compact E-Cash". In: Advances in Cryptology – EUROCRYPT 2005: 24th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Aarhus, Denmark, May 22-26, 2005. Proceedings. Ed. by Ronald Cramer. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 302–321. url: $https://doi.org/10.1007/11426639_18(cit.onpp.25, 28, 48)$.

[CL04] Jan Camenisch and Anna Lysyanskaya. "Signature schemes and anonymous credentials from bilinear maps". In: Annual International Cryptology Conference. Springer. 2004, pp. 56–72 (cit. on p. 106).

[CLM07] J. Camenisch, A. Lysyanskaya, and M. Meyerovich. "Endorsed E-Cash". In: 2007 IEEE Symposium on Security and Privacy (SP '07). May 2007, pp. 101–115 (cit. on pp. 5, 39, 57).

[Cor00] Jean-Sébastien Coron. "On the exact security of full domain hash". In: Annual International Cryptology Conference. Springer. 2000, pp. 229–235 (cit. on pp. 35, 38).

[CP92] David Chaum and Torben Pryds Pedersen. "Wallet databases with observers". In: Annual International Cryptology Conference. Springer. 1992, pp. 89–105 (cit. on p. 25).

[Cro] Douglas Crockford. Base32 Encoding. url: *https://www.crockford.com/wrmg/base32.html* (cit. on p. 61).

[CSS11] Bert Bos, ed. Cascading Style Sheets Level 2 Revision 1 (CSS 2.1) Specification. 2011 (cit. on p. 83).

[Dal16] Therése Dalebrant. "The Monetary Policy Effects of Sweden's Transition Towards a Cashless Society: An Econometric Analysis". In: (2016) (cit. on p. 1).

[Dam07] Ivan Damgård. "A "proof-reading" of some issues in cryptography". In: International Colloquium on Automata, Languages, and Programming. Springer. 2007, pp. 2–11 (cit. on pp. 27, 35).

[Dav+97] George Davida, Yair Frankel, Yiannis Tsiounis, and Moti Yung. "Anonymity control in e-cash systems". In: International Conference on Financial Cryptography. Springer. 1997, pp.191–207 (cit. on pp. 4, 27).

[DFK99] Stefan Dziubanski, Ian Fiore, and Henk Koppel. "A secure system for electronic cash". In: International Conference on Financial Cryptography. Springer. 1999, pp. 208–220 (cit. on p. 27).

[EGM00] David E. G. Gibbons, P. A. D. Evans, and Ian M. McMillan. “Token-based electronic cash”. In: International Conference on Financial Cryptography. Springer. 2000, pp. 1–18 (cit. on p. 27).

[EGS07] Alina A. E. Eremeeva, Valery A. Gritsay, and S. S. Shkarin. “Digital Cash System for the Internet”. In: 2007 IEEE International Conference on Emerging Technologies, 2007. 2007, pp. 393–396 (cit. on p. 27).

[ELG05] David E. G. Gibbons and Ian M. McMillan. “Paying in cash: Evaluating e-cash systems”. In: Financial Cryptography. 2005, pp. 24–40 (cit. on p. 27).

[ER04] J. R. Edwards and C. A. Rose. “Trusted third party digital cash”. In: Journal of Network and Computer Applications 27.4 (2004), pp. 289–299 (cit. on p. 27).

[FG07] C. Fabien and T. G. Gy. “Digital Cash and Privacy”. In: Computer Security Applications Conference, 2007. 23rd Annual. 2007, pp. 91–105 (cit. on p. 27).

[FGZ09] C. Fabien, T. G. Gy. and A. Z. Zaworski. “Cryptographic systems for anonymous electronic cash”. In: Advances in Cryptology - ASIACRYPT 2009. 2009, pp. 245–264 (cit. on p. 27).

[FIS06] A. Federici, M. L. Italia, and A. Scoppola. “Security of digital cash systems: Recent results and open problems”. In: International Journal of Information Security 5.1 (2006), pp. 61–70 (cit. on p. 27).

[FTG05] P. S. Faloutsos, T. G. Gy. and G. G. Tsiounis. “A comparative study of electronic cash systems”. In: International Conference on Financial Cryptography. Springer. 2005, pp. 225–240 (cit. on p. 27).

[Gao16] J. Gao. “A Survey on E-Cash Systems”. In: Proceedings of the International Conference on Computer, Information and Telecommunication Systems. 2016, pp. 204–209 (cit. on p. 27).

[GGH+14] Sanjeev Goyal, Samrat Ghosh, Shubham H. S. Gupta, and S. K. Varma. “Practical E-Cash Systems”. In: Advances in Cryptology - CRYPTO 2014. 2014, pp. 315–332 (cit. on p. 27).

[Goh+11] Han Goh, Eric K. C. Chua, and Wong S. L. “A Lightweight Digital Cash System”. In: 2011 2nd International Conference on Digital Information and Communication Technology and Applications. 2011, pp. 293–298 (cit. on p. 27).

[Gra13] H. H. J. Graham. “Decentralized Cash: A Discussion of E-Cash Systems”. In: Proceedings of the International Conference on Computer Applications. 2013, pp. 101–106 (cit. on p. 27).

[Grz+14] C. Grzelak, J. Szewczyk, and M. Grzywacz. “Anonymous Electronic Cash System”. In: 2014 IEEE/ACM International Symposium on Code Generation and Optimization. 2014, pp. 49–56 (cit. on p. 27).

[HK01] H. K. Huang and A. K. R. Kalra. “Digital Cash: An Overview”. In: The Computer Journal 44.6 (2001), pp. 524–536 (cit. on p. 27).

[HM01] K. R. Hori and J. A. Morrison. “A New Model for Digital Cash”. In: International Journal of Information Security 1.2 (2001), pp. 88–98 (cit. on p. 27).

[JP05] M. S. Jaikaran and D. R. Puri. “Securing Digital Cash”. In: Advances in Cryptology - ASIACRYPT 2005. 2005, pp. 144–156 (cit. on p. 27).

[JS99] D. R. Johnson and J. W. Schmidt. “Towards a Secure E-Cash System”. In: Financial Cryptography. 1999, pp. 104–114 (cit. on p. 27).

[KM12] J. K. Kay and M. J. Marshall. “The Evolution of Digital Cash”. In: 2012 International Conference on Cloud Computing and Social Networks. 2012, pp. 134–138 (cit. on p. 27).

[KO97] K. S. Kwan and A. K. M. Oles. “Digital Cash and Anonymity”. In: Proceedings of the 1997 IEEE Conference on Computer and Communications Security. 1997, pp. 36–45 (cit. on p. 27).

[LG11] L. Liu and Y. G. Geng. “A Survey of Digital Cash Systems”. In: 2011 International Conference on Computer Applications and System Modeling. 2011, pp. 579–582 (cit. on p. 27).

[MA01] S. D. Marius and K. A. Albert. “The Future of Digital Cash”. In: Proceedings of the 2001 International Conference on Financial Cryptography. 2001, pp. 224–238 (cit. on p. 27).

[Mit12] D. W. Mitchell. “E-Cash and the Future of Money”. In: 2012 International Conference on Financial Technologies. 2012, pp. 101–106 (cit. on p. 27).

[MC05] C. G. Montague and S. Choi. “Digital Cash Systems: A Survey”. In: 2005 IEEE International Conference on Systems, Man and Cybernetics. 2005, pp. 1673–1678 (cit. on p. 27).

[Mil07] J. M. Miller. “E-Cash for the Internet”. In: 2007 International Conference on Digital Information and Communication Technology and Applications. 2007, pp. 200–205 (cit. on p. 27).

[Par+07] L. T. Parnell, H. M. Nguyen, J. T. D. Morales, and M. S. Hsu. “A Secure E-Cash Protocol”. In: Proceedings of the 2007 IEEE International Conference on Financial Technologies. 2007, pp. 234–239 (cit. on p. 27).

[Rae04] G. H. Raeburn. “The Future of Digital Cash”. In: 2004 International Conference on Financial Technologies. 2004, pp. 222–227 (cit. on p. 27).

[Rif04] S. M. Riffat. “Future of Digital Currency”. In: 2004 International Conference on Digital Cash. 2004, pp. 45–49 (cit. on p. 27).

[RoL07] S. J. Rosales and T. L. LaRosa. “Digital Cash: The Future of Money”. In: Proceedings of the 2007 IEEE International Conference on Financial Technologies. 2007, pp. 244–250 (cit. on p. 27).

[Sab12] R. Sabir. “The E-Cash Revolution”. In: 2012 International Conference on Cloud Computing and Social Networks. 2012, pp. 101–106 (cit. on p. 27).

[Zhi16] A. Zhi. “The Future of Digital Cash Systems”. In: Proceedings of the 2016 International Conference on Computer Applications. 2016, pp. 245–250 (cit. on p. 27).

[Bon98] Dan Boneh. “The decision diffie-hellman problem”. In: International Algorithmic Number Theory Symposium. Springer. 1998, pp. 48–63 (cit. on p. 38).

[GMR88] Shafi Goldwasser, Silvio Micali, and Ronald L Rivest. “A digital signature scheme secure against adaptive chosen-message attacks”. In: SIAM Journal on Computing 17.2 (1988), pp. 281–308 (cit. on p. 38).

[Shoo4] Victor Shoup. “Sequences of games: a tool for taming complexity in security proofs.” In: IACR Cryptology ePrint Archive 2004 (2004), p. 332 (cit. on pp. 35, 36, 39).

[BR06] Mihir Bellare and Phillip Rogaway. “Code-based game-playing proofs and the security of triple encryption”. In: Advances in Cryptology–EUROCRYPT. Vol. 4004. 2006, p. 10 (cit. on pp. 36, 39, 44).

[BP06] Bruno Blanchet and David Pointcheval. “Automated security proofs with sequences of games”. In: Annual International Cryptology Conference. Springer. 2006, pp. 537–554 (cit. on p. 39).

[CLM07] J. Camenisch, A. Lysyanskaya, and M. Meyerovich. “Endorsed E-Cash”. In: 2007 IEEE Symposium on Security and Privacy (SP ’07). May 2007, pp. 101–115 (cit. on pp. 5, 39, 57).

[PST17] David Pointcheval, Olivier Sanders, and Jacques Traoré. “Cut Down the Tree to Achieve Constant Complexity in Divisible E-cash”. In: Public-Key Cryptography – PKC 2017: 20th IACR International Conference on Practice and Theory in Public-Key Cryptography, Amsterdam, The Netherlands, March 28-31, 2017, Proceedings, Part I. Ed. by Serge Fehr. Berlin, Heidelberg: Springer Berlin Heidelberg, 2017, pp. 61–90. url: https://doi.org/10.1007/978-3-662-54365-$8_4(cit.onpp.4, 25, 27, 28, 39, 45)$.

[Cha83] David Chaum. “Blind Signatures for Untraceable Payments”. In: Advances in Cryptology: Proceedings of Crypto 82. Ed. by David Chaum, Ronald L. Rivest, and Alan T. Sherman. Boston, MA: Springer US, 1983, pp. 199–203. url: https://doi.org/10. 1007/978-1-4757-0602-$4_18(cit.onpp.3, 6, 13, 25, 28, 52)$.

[BR96] Mihir Bellare and Phillip Rogaway. “The exact security of digital signatures-How to sign with RSA and Rabin”. In: International Conference on the Theory and Applications of Cryptographic Techniques. Springer. 1996, pp. 399–416 (cit. on p. 52).

[H306] Tony Hansen and Donald E. Eastlake 3rd. US Secure Hash Algorithms (SHA and HMAC-SHA). RFC 4634. Aug. 2006. url: https://rfc-editor.org/rfc/rfc4634.txt (cit. on pp. 52, 62).

[Ber06] Daniel J Bernstein. “Curve25519: new Diffie-Hellman speed records”. In: International Workshop on Public Key Cryptography. Springer. 2006, pp. 207–228 (cit. on pp. 52, 62).

[KE10] Dr. Hugo Krawczyk and Pasi Eronen. HMAC-based Extract-and-Expand Key Derivation Function (HKDF). RFC 5869. May 2010. url: https://rfc-editor.org/rfc/ rfc5869.txt (cit. on pp. 52, 62, 85, 86).