NTNU
Kunnskap for en bedre verden

Department of Engineering Cybernetics

TTK4550 - Engineering Cybernetics, Specialization Project

DefiFunds - A proof of concept Dapp built on Radix

Author:
Tobias Hagehei

Supervisor:
Sverre Hendseth

December, 2022

Abstract

In this project, a proof of concept Dapp (decentralized application) called DefiFunds has been made on the Radix network. It lets fund managers create decentralized funds and trade on behalf of users who deposit to the fund. Regular people can decide which funds they want to invest in and they are the only ones with access to withdraw their fund share. There is no need to trust a centralized third party to hold the tokens, as they are safely stored in a smart contract. A regular user will only need to find a fund manager that he thinks can trade profitably, there are no need to trust anyone apart from that. The Radix network is in its early days, and smart contracts are estimated to go live on mainnet in Q2 2023. This specialization project is serving as a starting point for a Dapp that hopefully can be launched when smart contracts goes live on mainnet.

This report starts with general knowledge about distributed ledger technologies and then goes into Radix specifically. Results showing how the Dapp is designed and implemented will then follow, and a demo will show how the Dapp works. Design, implementation, weaknesses and development of the project are then discussed. The report concludes with the proof of concept being a starting point for a fully functional Dapp. Finally, the tasks needed to complete the Dapp is listed.

Acronyms

BTC, ETH, USDT, USDC, DOGE Popular cryptocurrencies

Dapp Decentralized application

DeFi Decentralized Finance

DEX Decentralized Exchange

DLT Distributed Ledger Technology

LTV Loan To Value

NFT Non-Fungible Token

PoS Proof of Stake

PoW Proof of Work

Glossary

Arbitrage	A strategy that uses price differences between markets in order to make a profit.
Blueprint	In Scrypto the concept of smart contracts is split into blueprints and components. If we want to compare it to object-oriented programming, blueprints are classes, and components are objects. A blueprint contains all variables that will be part of the state of each instantiated component. The blueprint also holds the code for methods used to update the state of the component.
Burn	Indefinitely removing tokens from circulation.
Collateral	Something given to a lender as a guarantee of repayment.
Composability	Composability in the context of DeFi is the property of being able to make use of multiple smart contracts in one transaction.
Flash loan	A loan type unique for DeFi. Allows a loaner to take up as much loan as there is liquidity in a pool, use it for whatever it wants, and pay back the loan in the same transaction. If the amount is not paid back within the same transactions, all actions are reverted.
Mint	Creating or adding new tokens to circulation.
Overcollateralized	A loan type where the value of the collateral exceeds the value of the loan.
Pool	A common term in DeFi that is often used for liquidity pools, lending pools, mining pools, etc. It refers to liquidity pools in this report. A liquidity pool is a collection of tokens locked in a smart contract to facilitate trades.
Sybil attack	An attack on a computer network where a user tries to take over the network by operating multiple identities and undermine the authority in the reputation system.
Token	A digital unit that represents an asset or a utility. For example, a cryptocurrency or an NFT.
Whitelist	A list with trustworthy addresses.

Table of Contents

Introduction

The goal of this project is to make a proof of concept Dapp (decentralized application) where traders can create decentralized funds, and everyday people can invest in them. Traders will effectively show if they can trade profitably and do not need to go into the complex process of creating a centralized hedge fund to get everyday people to buy their funds. In addition, regular users will have the convenience of being able to invest in the decentralized finance ecosystem without doing the trading themselves or going to a centralized platform.

If you know the basics about blockchain and Ethereum, you can most likely skip the background chapter about distributed ledger technology, but I would recommend looking at the scalability problem subsection to understand my motivation better. If you know basics about Radix and just want to understand the blueprints, you can go straight to the results. I would recommend doing the demo to get a better understanding of how the Dapp work.

1.1 Motivation

My main motivation for this specialization project comes from my strong interest in cryptocurrencies and distributed ledger technologies since 2018. I really like the concept of being able to do financial transactions without relying on centralized institutions, mainly because of the trustless nature of smart contracts and the efficiency it creates. I have used lots of decentralized applications in the past and thought it would be cool to build one myself.

As an active user of the Ethereum network, I have seen its scalability problems, with only 30 transactions per second on average. In addition, the fees paid for getting a transaction through have been in the hundreds of dollars because of the competitive fee structure. This has made me look into several other networks where Radix, in my opinion, looks promising with its potential linear scalability without losing composability.

From a developer perspective, Radix also has a promising language for writing smart contracts. The language is called Scrypto and is built on rust. It is designed specifically for moving resources around, and the term used to describe it is "asset-oriented." In my opinion, it is much faster and simpler to write bug-free code on Radix than on Ethereum.

The reason for wanting to build this Dapp is that I think it has huge potential, and that there exist few to no good solutions for it yet. If you want to invest in the cryptocurrency market, you either have to buy and hold, do the trading yourself, or go to a centralized website. One Dapp called dHedge has got some traction doing what I am trying to build but has had little success so far [16]. My guess is that people aren't using it because of the exceptionally high fees associated with that kind of Dapp on Ethereum. At least that was my reason for not using it when I first discovered it.

I figured out I wanted to build this Dapp on Radix for the two main advantages mentioned above: potentially linear scalability without losing composability and the asset-oriented language. Radix is still in its early stages and hasn't put smart contracts onto mainnet. The possibility of Radix not delivering what it's planning to deliver and not getting mass adoption is huge, but I think it's worth taking a look at it and building a proof of concept Dapp.

Background: Distributed ledger technology

This chapter is written to give general knowledge about distributed ledger technology (DLT). Section 2.1 explains what a DLT is and introduces some examples of existing DLTs. Section 2.2 explains what a smart contract is and some associated challenges. Section 2.3 gives some examples of what these smart contracts are used for in decentralized finance (DeFi) and introduces a problem currently affecting the adoption rate of DeFi. Both Section 2.2 and Section 2.3 are written mostly with Ethereum in mind since Ethereum is the largest DLT with substantial smart contract functionality. Terms more specifically used for Radix are described in chapter 3.

2.1 Distributed ledger technology

Distributed ledger technology has become quite popular in the recent years. Primarily because of the technology later named blockchain, which was introduced in 2008 [31] as a part of a proposal for Bitcoin. Blockchain has become a relatively known technology in recent years and is often misused as the word for DLT. Other structures used as a DLT for cryptocurrencies are for example a directed acyclic graph; Iota is one of them[4]. There also exist other types of DLTs, and the Radix network which is going to be the main focus of this project is one of them.

2.1.1 Database

Storing and managing data is an important part of most applications today, and a database is an important tool for making that happen. A database is a collection of structured data that is often stored electronically. It is often modeled in rows and columns in a series of tables and is used for having easy access to data. The type of operations that can be performed on a database varies depending on the database type, and some basic examples of operations are, create, read, update, and delete. Three types of databases relevant for understanding a DLT are centralized, decentralized, and distributed databases [34].

A centralized database is stored in a single location and maintained by a single node. The main advantages of a centralized database are that it is easy to access, easy to coordinate data, and cheap to maintain since all data is stored in the same place. However, a centralized database has drawbacks in performance, redundancy, and availability. If the system is down for a while or fails totally, there will be no access to the data, and the data itself can be destroyed.

Figure 2.1: Visualizations of different database structures [3]

Decentralized databases do not have central storage. The entire data is spread across multiple servers, often in different physical locations, and managed by several nodes. When a node modifies the data, it is reflected on all the other nodes. In contrast to a centralized database, decentralized databases are more redundant, but it is a bit harder to coordinate the data. Nodes are organized in a hierarchical structure where a set of nodes communicate with a specific set of nodes that contains all the data. A decentralized database can be seen as several smaller centralized databases where a superordinate set of nodes coordinate on having the same data.

A distributed database is similar to a decentralized one in that it also is stored in different physical locations. A distributed database is usually not organized in a hierarchical structure and often forms a mesh, as illustrated in Figure 2.1. All nodes have equal rights for changing the data in the network, which can lead to different nodes having different records of the database. Even though a distributed database is architecturally distributed, it is considered logically centralized, which means that operations performed on a node should lead to the same results independent of which node the operations were performed on. The process of reaching consensus on a distributed system is generally harder on a distributed system than on a decentralized one, since all nodes are part of the data storing. To reach consensus across the network, a consensus mechanism is used which will be more detailed explained in Section 2.1.3. A huge advantage of a distributed system is that it is far more reliable. If one node goes down, the network will still be able to communicate; the traffic will just go through other nodes. Since nodes can work independently, the network itself is generally more scalable and efficient.

2.1.2 Distributed ledger

A distributed ledger is a special type of distributed database [34], where only append and read operations are valid. If a person would want to change a value in the ledger, an append operation is done, and the old value is stored on the ledger. A ledger keeps track of all updates and stores them chronologically. A ledger is also tamperproof so that everyone can cryptographically check that the data has not been tampered with. Ledgers presume the existence of malicious nodes and use a consensus mechanism to achieve consistency across the network. DLT allows distributed ledgers to be run on arbitrary nodes where providers do not need to be trusted or known. The nodes can also arbitrarily join or leave the network without affecting consistency.

2.1.3 Consensus mechanism

To ensure that all nodes agree on a single database record, a consensus mechanism is used [13]. It is a crucial part of distributed ledger technology since no central authority can control the network or the data stored in it. For simplicity's sake proof of work (PoW) and proof of stake (PoS) are often referred to as consensus mechanisms, but in reality, it is only a part of the consensus mechanism.

PoW and PoS are the most common Sybil resistance mechanisms used in distributed ledger technology [13]. A Sybil attack is an attack where a user tries to take over a network by pretending to be many users or nodes. Since a DLT is a decentralized network with no input from a central authority, Sybil resistance is essential. In PoW a Sybil attack is minimized by getting users to spend a lot of CPU power and in PoS by letting users put their tokens up as collateral. A user will then have a weighted vote based on their CPU power used or collateral put up.

In this subsection, three different consensus mechanism is introduced; A PoW-based consensus mechanism often named Nakamoto consensus, a PoS-based consensus mechanism called Gasper and Cerberus which also uses PoS for Sybil resistance.

Nakamoto consensus - Proof of work based

Proof of work as a concept was introduced in 1993 by Cynthia Dwork and Moni Naor, and later applied by Satoshi Nakamoto in 2009 when Bitcoin was invented [24]. Nakamoto also came out with a whitepaper in 2008 where PoW and the technology in general are explained [31]. PoW is a cryptographic proof used to show that a certain amount of work has been done. On the Bitcoin network, PoW involves hashing nonces using the sha256 algorithm, searching for a hashed value with the required 0 bits. The average work required is exponential to the required zero bits and is dynamically adjusted based on the total work of all computers in the network. Once a hashed value with the required zero bits has been found, a block gets added to the blockchain, and it can't be changed without doing all the work again. If someone should want to change a block from the past, he needs to redo the work for all the blocks after it.

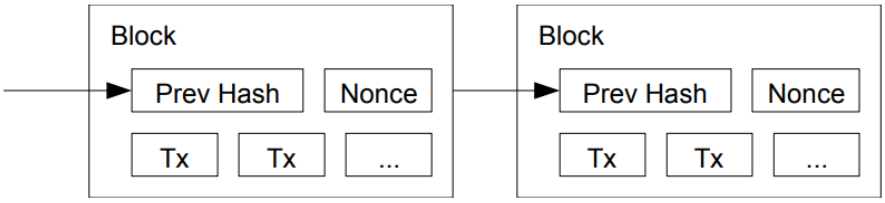

Figure 2.2: Chained blocks [31]

Another part of the consensus mechanism in Nakamoto Consensus is the longest-chain rule. The longest blockchain is considered the valid blockchain and is statistically going to be the chain with the majority of the work on it. If honest nodes then control more than 50% of the CPU usage, it controls the blockchain. If an attacker with less than 50% of the CPU power tries to attack the network, the probability of the attacker catching up to honest nodes diminishes exponentially as new blocks get added to the blockchain.

After one node has added a block to the blockchain, other nodes will update their record to include that block as well. If two nodes find a new block almost simultaneously, different nodes will have different records of what block is added. They will also save the other block in a branch in case that chain gets longer. Nodes will continue to work from the block they first saw, but will switch over if a new block gets added to the other.

Mining is a term used to describe the process of creating new blocks on the blockchain. When someone finds the hashed value with the required 0 bits, a new block is created, and a reward for finding the value is given to the node that found it. This technology gives incentives for nodes to mine and keeps the networks secure. A disadvantage of this feature is that it competes on CPU and energy usage. This has led to an estimated energy consumption of 130 TWh as of October 2022, which is comparable to the electricity usage of Argentina [7].

Gasper - Proof of stake based

Proof of stake is an alternative Sybil resistance mechanism and has been used on Ethereum since September 2022 [33]. Before that, Ethereum also used the Nakamoto consensus. Instead of using CPU power as the voting mechanism, Ethereum is using staked ether. In other words, ether that is locked up as collateral. If a node votes the same as the majority, it will be rewarded, but if it votes against it, it will be punished. A node with voting rights on Ethereum is called a validator and needs to hold 32 Ether as collateral. The network picks a random validator to propose a block, and the other validators votes for or against this block to be added to the blockchain.

For incentivizing people to stake and secure the network, there is something called staking reward [23]. This reward is adjusted based on how much ether is staked and consists of transaction fees from users and issuance of new ether from the Ethereum network itself.

Cerberus

Cerberus is the consensus mechanism that will be used by the Radix network in its final form. Radix uses a form of PoS as the Sybil resistance mechanism, called delegated PoS [19]. In delegated PoS, the users do not participate in the voting but rather give the voting power to a validator or several validators it trusts. This makes it easier for people to stake and generally increases network security since more tokens are needed for a Sybil attack. On Ethereum, each staker needs to run their own validator to be able to stake. However, several centralized solutions exists that let people stake Ethereum through them.

The consensus part of Cerberus is a lot more complex than Gasper and Nakamoto Consensus and will not be explained here, but can be read in detail in the Cerberus whitepaper [10], or in simpler form in the infographic series [37]. The main advantage of Cerberus is that it achieves unlimited linear scalability while still preserving atomic composability. Ethereum also plans to increase its scalability with sharding, but it is up to debate how composable it gets [14].

2.2 Smart contracts

2.2.1 What is a smart contract?

A smart contract is a contract that is automatically executed when certain conditions are met, and is typically stored on top of a DLT. There exists smart contracts on Bitcoin, but only simpler ones, since Bitcoin is Turing incomplete. On the other hand, Ethereum is Turing complete and has been the platform with the most smart contract development on. [26]. One can divide a smart contract into four phases to better understand how it works: creation, deployment, execution, and completion [44].

The first step is the creation of the smart contract. In this step, the parties involved in the deal must agree on certain parameters. A simple example can be a trade between two cryptocurrencies. One receives x amount of dollars, and the other part receives y amount of ether. After the deal is agreed on, the code needs to be developed.

After the smart contract is created, it needs to be deployed on top of a DLT. Since a DLT is immutable, the smart contract can't be modified after the deployment. If parts of the contract need to be modified, the contract needs to be redeployed. The old contract will still exist on the DLT. The code is often already created and deployed, so normal users can normally skip this and the previous step.

Once a smart contract is deployed on a DLT, all users of the network can call the contract. If all criterias for the smart contract are met, the smart contract executes. When a smart contract is called, transactions with all state changes defined by the smart contract are sent to the network and validated by the nodes on the network. A smart contract is also deterministic, meaning that

the outcome of the contract will be the same for everyone who calls it.

After the transactions have been executed, new state changes according to the smart contract are updated. All transactions involved in the process and the new state changes are stored on the DLT. The cryptocurrencies from the trade have switched hands, and there was no need for a middleman.

Once a smart contract is deployed to a network, no third party controls the smart contract. Even though individuals or organizations create them, they have no exclusive right over them. An important part of a smart contract is that there is no need to trust the third party that developed the smart contract to use it. It can be verified by yourself or other parties you trust. Code is law, and there is no way of changing a smart contract once deployed.

2.2.2 Security in smart contracts

Even though smart contracts are secure in the way that no third parties can override them, there are still several security problems. The two main problems are that it's hard to write bug-free code and that it's hard to verify for the user what transaction he is signing. In a centralized banking system, the bank can change balances or other fields in a database if bugs should occur. In a decentralized system, it is not possible [44]. We will go through some examples to illustrate the problems.

```
contract WalletLibrary {
    address owner;

    // called by constructor
    function initWallet(address _owner) {
        owner = _owner;
        // ... more setup ...
    }

    function changeOwner(address _new_owner) external {
        if (msg.sender == owner) {
            owner = _new_owner;
        }
    }

    function () payable {
        // ... receive money, log events, ...
    }

    function withdraw(uint amount) external returns (bool success) {
        if (msg.sender == owner) {
            return owner.send(amount);
        } else {
            return false;
        }
    }
}
```

Figure 2.3: Bug in a smart contract [8]

Vulnerable wallet example

Figure 2.3 is a snippet of a simplified smart contract written in Solidity. It has a vulnerable bug and goes under the name "Party MulitSig Bug" [8]. This Particular bug was discovered by white hat hackers and drained around 78 million dollars worth of cryptocurrencies from the contract. The code snippet illustrated is a simplified version of the real contract, but is good for understanding the point. The first function sets the owner of the wallet and a lot of other parameters. The second function is a function that lets the current owner set a new owner. The fourth function is a function that lets the owner withdraw the funds. On Ethereum the default visibility for functions are public, so everyone can call them if not specified otherwise. The problem with the first function was that it was believed to only be callable once, but it could actually be called by everyone. Luckily a group of white hat hackers discovered this bug first, set themselves to owners, and withdrew all the funds.

Signing smartcontracts

It may sound strange, but a common practice on Ethereum has become blind signing transactions. Blind signing means confirming transactions you do not know the full details of [29]. One of the reasons may be the quickly evolving space. The UX has not kept up with the smart contract development and has led users needing to trust the smart contract instead of verifying it themself. When smart contracts first came, there was no way to go around blind signing if you wanted to use simple Dapps like Uniswap. Uniswap is the largest decentralized exchange by trading volume as of December 2022 [21]. The problem with many smart contracts is that the important contract details, like what amounts should be sent and received, can not be fully extracted from the contract in a user-readable language. The user interface can often display something like "data present" or a long hash value, instead of key details needed to verify what is signed.

Developers have in general focused on "ease of use", instead of secure implementations. When doing a token swap on Uniswap a user needs to confirm two transactions. The first transaction is a token allowance transaction that lets the smart contract spend the token. The other transaction is a transaction that executes the token swap. A common practice has been to give unlimited token allowance, meaning that users give the smart contract rights to spend an unlimited amount of that token on their behalf. In daily life, it can be compared to an automated bill from an electricity company. The company can charge you want they want. Still, they have little incentive to charge a higher amount than your bill was supposed to be. This will be categorized as illegal, and the bank can easily reverse the action for you [12]. Once a payment is made with smart contracts, the action is irreversible. Hackers have higher incentives here since is harder to get caught by governments, and the transaction can't be reversed by a third party.

Phishing

On a DLT, users have full responsibility for the funds themselves, which has led to lots of phishing scams. There are not that many reports on phishing scams related to smart contracts in general, but a report covering Non-Fungible Tokens (NFTs) shows that phishing scam is the most common scam type [6]. There are two main ways phishing attacks occur:

The first and most common attack type is through a fake popup that looks like a real custodial wallet, like Metamask. The provider typically asks users the re-enter their private key. Once entered the scammer has access to their wallet and can drain all NFTs and cryptocurrencies without any third party stopping the action.

The second most common attack type is encouraging users to sign malicious smart contracts. The main problem here is that blind signing is a common way of signing. Scammers typically make use of the unlimited token allowance function for fungible tokens or similar functions made for non-fungible tokens.

2.2.3 Oracle problem

Connecting real-world applications to a smart contract is quite challenging and is called the oracle problem [11]. A DLT only need to form consensus on binary data already stored on the DLT. The problem with connecting real-world data to a DLT is that it is hard to determine what data is correct. Take for example the price of ether. It can be one price on a specific exchange and another on a different exchange. The problem here is what source you should trust.

The main reason for wanting a smart contract is to achieve deterministic output based on the inputs. If the inputs can vary based on the different data sources, there is little use for a smart contract. Getting information from a trusted centralized third party would be more secure than getting information from a random data source. However, the trusted centralized data source still suffers from the same problems as centralized databases, like corruption and downtime. Smart contracts relying on an oracle are no more secure than the oracle itself.

Chainlink is the most used oracle service as of today [15]. It currently supports 14 platforms where some of them are Ethereum, BNB chain, and Solana. Chainlink has several features, and one of them is price feeds. It lets smart contracts retrieve the latest pricing data of a token. Each price feed is updated by multiple independent oracle operators. The number of oracles contributing to each price feed varies, and some price feeds are therefore more secure than others. The provided prices are validated and aggregated by a smart contract and form a final answer. For an update to take place, a minimum number of oracle nodes need to provide their price. The minimum number varies from price feed to price feed.

2.3 Decentralized finance

Decentralized finance, commonly known as DeFi, uses DLTs to manage financial transactions such as transferring, lending, borrowing, and trading. Today most of the financial transactions are done by centralized systems, but decentralized systems have started to take a little part of the financial sector. In this section, we will go through some examples of where DeFi is used today and some of its limitations.

2.3.1 Decentralized finance usecases

Self custody

Self-custody is the standard way to manage your assets in DeFi. Self-custody refers to users being in absolute control of their assets [32]. Their assets are backed by a private key and often stored in non-custodial wallets like Metamask or Ledger [30]. In traditional finance, it can be compared with holding your assets in cash or physical gold. DeFi's advantage regarding self-custody compared to centralized finance is that you still can use financial services, like sending money abroad, taking up a loan, and trading stocks.

In centralized finance, a third party controls your funds, usually a bank or an asset manager. If they were to go bankrupt, lots of your assets could get lost depending on your wealth and insurance. In 2013 users of the Bank of Cyprus lost up to 60% of their total holdings for deposits exceeding 100,000 EUR [34]. A bank can also choose to confiscate your funds or not let you send money to those you want. With DeFi and self-custody, these problems are eliminated as no third party or authority has access to your private key or funds.

The paradox with not having a third party to control your assets is that no one can access your funds if needed. If you lose your private key or password, there is no way to recover the funds. If you forget your password to the bank, you can likely go to the bank and ask for a new password.

Lending

In DeFi the two most common loan types are overcollateralized loans and flash loans [36]. The more commonly used in traditional finance, undercollateralized loan, is almost non-existent in DeFi due to handling defaults of loans. In traditional finance, an undercollateralized loan is dependent on a government and a debt collection agency to handle defaults of loans. In DeFi, there is currently no good implementation of this loan type, and we will focus on the two other loan types in this section. A flash loan can be fully executed on a DLT without any third party, or oracle. Overcollaterlized loans make use of an oracle, typically Chainlink, to get the price of different assets. Apart from that, overcollateralized loans do not need any oracle or third party.

An overcollateralized loan, is a loan where you put up more collateral than you take off loan. An example can be to put up 1000 dollars worth of ETH in collateral and take up a loan of 500 dollars worth of USDC. You now have a loan-to-value (LTV) ratio of 50% [5]. As the collateral fluctuates in price, the LTV ratio adjusts. If the LTV ratio gets to close to 100%, the risk of

getting undercollateralized is there. To avoid this scenario, there is a liquidation threshold. When the liquidation threshold is met, any actor can liquidate you by buying part of your collateral at a discounted price.

A flash loan is a loan where you do not need to put up collateral, and you can take up as much loan as there is capital in the smart contract [5]. The loan is given and paid back within the same transaction. If a loan taker fails to pay back his loan within the same transaction plus interest, the loan is reverted. This is possible due to the atomic nature of smart contracts. Flash loans are not used in traditional finance, but play an important role in DeFi. We will explain how a flash loan works through a real arbitrage example that has happened on Ethereum.

Figure 2.4: An example of a flash loan used to do an arbitrage trade.
transactions hash: 0x01afae47b0c98731b5d20c776e58bd8ce5c2c89ed4bd3f8727fad3ebf32e9481 [1]

1. A user started by taking up a flash loan of 405k USDC (He already had 45k USDC).
2. He then exchanges 450k USDC for 493k USDT on Uniswap.
3. He exchanges 493k USDT for 493k USDC on Curve.
4. He repays the 405k of USDC and keeps the remaining 43k USDC in profit.

All these actions took place in the same transaction. If the borrower would not be able to pay back the USDC he borrowed, all the actions in the transaction would be reverted. An example on a reverted flash loan transaction can be found here: [2]. Flashloans have several other use cases too, like for example liquidating an overcollateralized loan that has reached its liquidation threshold.

Decentralized exchanges

Decentralized exchanges (DEXes) have had a rapid growth curve over the last few years. In January 2020, the monthly DEX volume was around 1 billion dollars, and two years later in January 2022 the monthly trading volume had risen to about 120 billion dollars [21]. A DEX is one of the better ways to show where smart contracts make a difference. Transaction costs are cut to a fraction, there are no restrictions on what cryptocurrencies can be traded, and they are available to use at any time.

There exist several types of decentralized exchanges. Some are order book based, like mostly done on centralized exchanges, while others use an approach with bonding curves. A simple but effective exchange method is the constant product automated market maker used by Uniswap [25]. Uniswap is the most popular DEX by trading volume as of December 2022 and has been it for a couple of years [21]. We will take a closer look at how Uniswap V2 works.

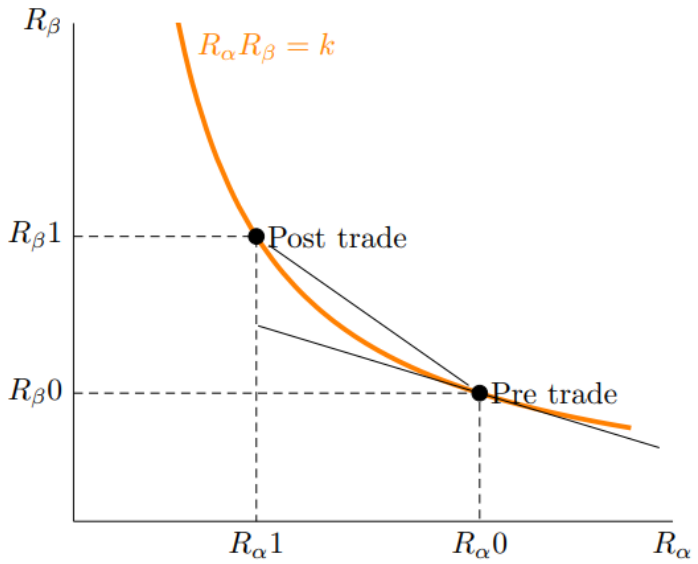

Figure 2.5: A constant product automated market maker [25]

A constant product automated market maker follows the simple function $R_\alpha R_\beta = k$. R_α is the reserve of asset α and R_β of the asset β. The product of them will be k before and after trades are done. When someone wants to do a trade, a transaction with x amount of α is sent to the smart contract containing the constant product function. The smart contract calculates the output amount of β based on the constant product function and sends the calculated amount of β in return. In an automated marker, the trades are made between market takers and liquidity providers. Market makers are buying or selling the asset, and liquidity providers provide liquidity to the pair. When providing liquidity to a pair, you send α and β to the smart contract in the same ratio as the total liquidity in the smart contract. To incentives people to provide liquidity to the smart contract, they receive a small fee from each trade.

2.3.2 Scalability problem

There has been, and still is, a problem with DLTs not being scalable enough to meet the high demand. The average transactions per second is about 15 on Ethereum as of 2022 [18]. Users of the Ethereum network compete to be among those 15 transactions by overpaying each other on fees, which has led to a very high cost for using the network. The average transaction cost for using Ethereum was in the range of 20-50$ in q4 2021 [17]. This has led to average users not being able to use the network. In response to the high price of using the network, an alternative DLT called Binance smart chain was made. It was able to handle a lot more transactions per second but met criticism for not being decentralized enough. It only has 21 nodes validating the network and was considered a centralized solution by many [28]. In this subsection, we will further look into the scalability problem on blockchains and possible solutions for it.

The Scalability Trilemma

Vitalik Buterin, the founder of Ethereum, states that there is a scalability trilemma where you can only get two out of three properties on a blockchain, if you stick to "simple" techniques. The three properties are scalable, decentralized, and secure [9]. The scalability trilemma is a hypothesis thought to be true by early developers, but hasn't any proof for or against it. The properties are defined as follows:

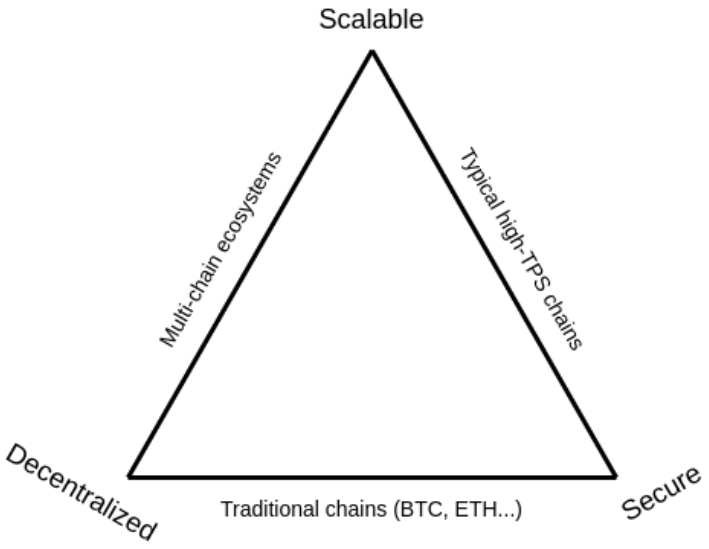

Figure 2.6: The Scalability Trilemma [9]

- Scalable: The blockchain can process more transactions than a single ordinary node. How scalable the blockchain is, is often measured in how many transactions it can handle per second.
- Decentralized: The blockchain can run without trusting a small group of centralized actors. The more nodes, and the less power they have individually, the more decentralized.
- Secure: The blockchain can resist a large number of malicious nodes participating in the consensus mechanism. Ideally, 49% of the nodes can act dishonestly, and the network still not be corrupt.

Sharding

Sharding is one of the solutions for making blockchains more scalable while still holding the decentralized and secure property from the Scalability trilemma. It is also the primary method protocol developers on Ethereum are working on. In simple terms, sharding means splitting up the verification process horizontally so that only a subset of all nodes verifies a block in the blockchain. This is done to spread the load so that the overall network can handle more transactions [9].

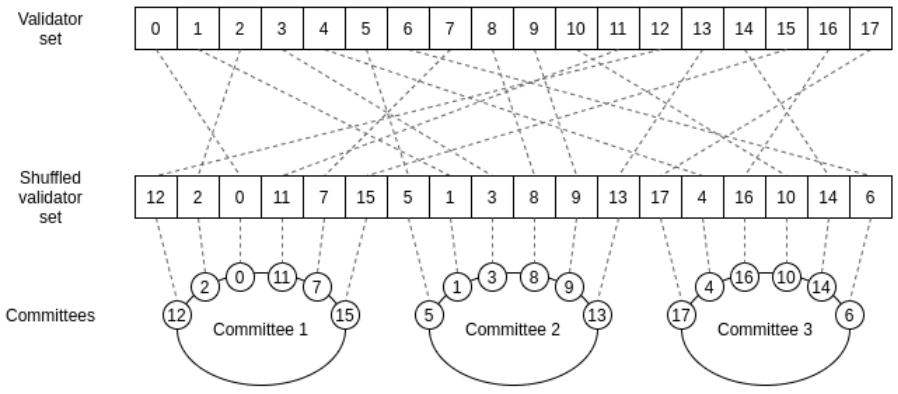

Figure 2.7: Sharding through random sampling [9]

We will go through random sampling, a simplified version of the sharding method that will be implemented on Ethereum [9]. See Figure 2.7. Say we have 18 validators and three blocks that need to be verified before the next set of blocks is incoming. We start by shuffling the validator list and randomly selecting a committee of validators to verify a block. Each validator in the committee verifies the block and publishes a signature as proof that they verified it. A validator only verifies the single block it's assigned. It also needs to verify the signature of the two other blocks. Verifying a signature is a lot faster than verifying a hole block. The reason for shuffling the validator list before each set of blocks is to prevent malicious actors from easily corrupting a committee. They now need to control close to a majority of all validators, instead of only a majority of the validators in a committee.

Composability

Sharding on Ethereum is thought to meet all three properties on the scalability trilemma, but it's up for debate how composable it gets after its sharding [14]. Composability in the context of DeFi is the property of being able to make use of multiple smart contracts in one transaction [20]. The reason that composability is important is that lots of Dapps make use of it today. The clearest example is the use of flash loans. Between the loan and the payback, the user makes use of several smart contracts. If those smart contracts were on different shards, the atomicity of the whole flash loan would not be possible, and if a flash loan can't be atomic, a flash loan can't exist.

On Ethereum, the composability question is still up for debate, but Radix claims to have solved the trilemma and the composability problem using its unique upcoming consensus mechanism called Cerberus [10]. The claims made by Radix are also verified in another independent paper published by the University of California Davis [22].

Background: Developing on Radix

3.1 Radix and the Radix Engine

Radix is a DLT that is made to handle the problems that DeFi currently is facing [27]. Radix claims to have solved the scalability trilemma without breaking atomic composability. It has also made its own language called Scrypto as an answer to the complex process of developing Dapps on DLTs like Ethereum.

The Radix Engine is the application layer on Radix and is similar to the Ethereum virtual machine on Ethereum. It is built to work hand in hand with Radix's future consensus layer called Cerberus. The main difference between The Radix Engine and the Ethereum virtual machine is that the Radix Engine is built around an asset-oriented approach, and uses a finite state machine to manage tokens. The Radix Engine makes sure that tokens never can be drained or lost and gives tokens a similar behavior as physical coins.

3.2 Scrypto

Scrypto is a programming language based on Rust. It is designed to develop Dapps with asset-oriented features and is the native langue to Radix. In this section, we will take a look at important parts of the language. Most of the information in this section is taken from the Scrypto 101 course [27] and the Radix docs [41] [42].

In Scrypto the concept of smart contracts is split into blueprints and components. If we want to compare it to object-oriented programming, blueprints are classes, and components are objects. A blueprint contains all variables that will be part of the state of each instantiated component. The blueprint also holds the code for methods used to update the state of the component.

3.2.1 Resources

On the Ethereum network, tokens are implemented by developers as smart contracts. Two of the most common token standards on Ethereum are ERC-20 tokens which are a standard for fungible tokens, and ERC-721 which is a standard for non-fungible tokens [35]. The token standard has its own set of rules and methods related to it. Sending a token on the Ethereum network actually means calling a method on a smart contract that someone deployed. A token contract is simply a mapping of balances to addresses.

On the Radix network tokens, also referred to as resources, are natively understood by the platform. The whole Scrypto language is designed around handling resources in a safe way and is taken care of by the Radix Engine itself. The developer can then think of a token transfer as actually sending a token.

Figure 3.1 is an example of how a token is defined in Scrypto. It contains flags telling what metadata is connected to it, if it is burnable and mintable, what the initial supply is, and how many subparts a token can be divided into. The "rule!" function defines who can mint and burn the token and will be explained more about in Section 3.2.2.

```
let share_tokens: Bucket = ResourceBuilder::new_fungible()
    .divisibility(DIVISIBILITY_MAXIMUM)
    .metadata("name", "share tokens")
    .metadata("description", "Tokens used to show what share of the fund you have")
    .mintable(rule!(require(internal_fund_badge.resource_address())),LOCKED)
    .burnable(rule!(require(internal_fund_badge.resource_address())),LOCKED)
    .initial_supply(initial_supply_share_tokens);
```

Figure 3.1: A resource defined in scrypto

Resource container

Buckets and vaults are the two resource containers that exists on the Radix Engine. They are relatively similar and both of them can only hold one type of resource. The main difference between buckets and vaults is that buckets are transient. That simply means that buckets only exists within the length of the transaction itself. Vaults on the other hand are permanent resource containers. When a transaction has finished, the resources that still are in buckets need to be put in a permanent resource container, or else the transaction will fail.

Finite State Machine

Resources in Radix are implemented through a finite state machine; see Figure 3.2. When a resource is first minted, it is put into a bucket. A resource can then be moved to a vault and at a later stage taken from that vault. A resource always needs to be in a bucket or a vault, and it can never be in a bucket and a vault at the same time. In this model, we can't have dangling resources. So if a resource is not put into a bucket or a vault, it needs to be burned. The finite state machine ensures that these principles are followed and returns an error if a developer tries to do otherwise.

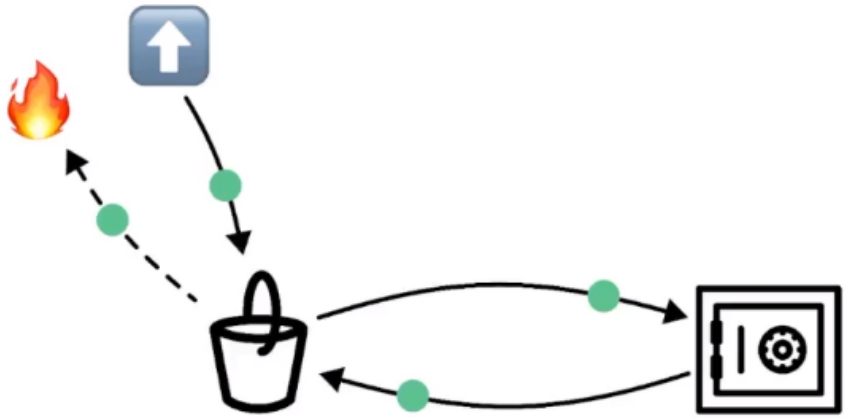

Figure 3.2: Finite State Machine [27]

The reason for having a finite state machine is to make tokens intuitive to understand and easier to avoid critical bugs when developing. Resources are more reliable, and states which are not thought about yet are impossible. Finally, this makes it safer and faster to develop new Dapps.

3.2.2 Authorization

In a smart contract, all methods are permissionless by default. This means that by default everyone can call a public method on a component. To restrict this access, some kind of authorization is needed. Scrypto's authorization model starts with three core concepts: badges, proofs, and access rules.

Badge

A badge is a resource used to gain access to restricted methods or actions. It is not a special type of resource; it is just a regular resource and is considered as a badge if it is used for authorization. On other networks such as Ethereum, the normal way to do authorization is through an address-based model. When you then want to restrict access to a method you verify that the caller's address is equal to the address needed to call that method. On Radix, a badge is used for authorization. If you would like to change an admin you can simply send that badge to the new admin. The idea of a badge is designed around the assets-oriented approach and is a more flexible way to restrict access.

Proof

When you want to show that you own a specific badge, you create a proof of the badge and send the proof. Proofs are a way to show that you own a specific badge without sending the badge itself. If you send the badge instead of the proof, it is the same as sending over the access rights. Proofs are a copy of the badge and only exists within the duration of a transaction. A proof can be created by all types of resources and can be used to show that you own specific tokens.

Access Rules

Access rules can be set on several actions on resources like shown in Figure 3.1. Here a specific badge is required to mint or burn the resource. There exist several other access rules, and some of them are "allow_all", "deny_all", "require_any_of", and "require_amount". You can also combine several of these access rules together by using logical operators.

Access rules can also restrict access to certain methods on a component. By default, all public methods are accessible by everyone, but you can define access rules for public methods on a component like shown in Figure 3.3. To be able to call a method with an access rule you need to show a proof of the required badge.

```
// Initialize struct and instantiate as a local component
let local_component: LocalComponent = Self {
    some_data: some_initial_value
}
.instantiate();

// Define access_rules
let access_rules = AccessRules::new()
.method("change_deposit_fee_fund_manager", rule!(require(fund_manager_badge.resource_address())))
.method("withdraw_collected_fee_fund_manager", rule!(require(fund_manager_badge.resource_address())))
.method("trade_radiswap", rule!(require(fund_manager_badge.resource_address())))
.default(rule!(allow_all));

// Apply access_rules, and add component to the global address space
let component_address = my_component.add_access_check(access_rules).globalize();
```

Figure 3.3: Access rules

3.3 Current state of Radix and Scrypto

The Radix public network is live, and has been live since July 2021 [38]. Token creation and token transfers are possible, but smart contracts are not live as of writing. Babylon, a major milestone release, is estimated to come in Q2 2023. It will enable smart contracts and bring all scrypto features to the Radix public network. Babylon Alphanet went live in the late part of Q3 2022, and Babylon Betanet is estimated to come within the end of 2022. The alphanet was the earliest possible public test network with capabilities similar to the Radix Engine simulator, and the betanet will be an upgraded version of the test network.

Scrypto is also under constant upgrades and, as of writing in v0.6. Scrypto has also been in v0.5 during this specialization project. There were only minor changes to make it work with alphanet, and it didn't cause a problem under development.

Results

4.1 Requirements

This requirement specification gives a brief summary of what is required of the Dapp. The general idea of the Dapp and my motivation are described in chapter 1. Requirements were selected based on how to keep the user funds as secure as possible while still allowing the Dapp to work as intended. Here is the list with requirements for the proof of concept Dapp:

- A user should be able to deposit and withdraw from a fund without relying on a third party.
- No one should have access to get tokens from the fund directly or indirectly apart from when users withdraw their share of the fund.
- The tokens in the fund should only be tradable by the fund manager.
- There should exist a way for the creator of the Dapp and the fund managers to collect a fee for their work.
- The user should not receive less of his share than he originally anticipated when he joined the fund, because of fee changes.

4.2 Design and Implementation

The proof of concept Dapp I have made consists of three blueprints: Fund, DefiFunds, and Radiswap. I have made Fund and DefiFunds, while the Radiswap blueprint is made by members of the Radix team [40]. Radiswap is a simple example of an automated market maker and will likely not be the blueprint I will use in the final version of this Dapp. However, it is good to use in a proof of concept before a final version of an automated market maker is made. In this section, I will not go into detail on how the Radiswap blueprint works, as it is not made by me. However, I will explain the most important parts of the two other blueprints. You can find the code files for the blueprints in Appendix A.

Figure 4.1 shows how the blueprints work together. The fund blueprint makes use of some methods from DefiFunds and depends on "whitelisted_pool_addresses" and "defifunds_deposit_fee". Fund also makes use of the swap method from the Radiswap blueprint. The DefiFunds blueprint holds a list with all the funds created.

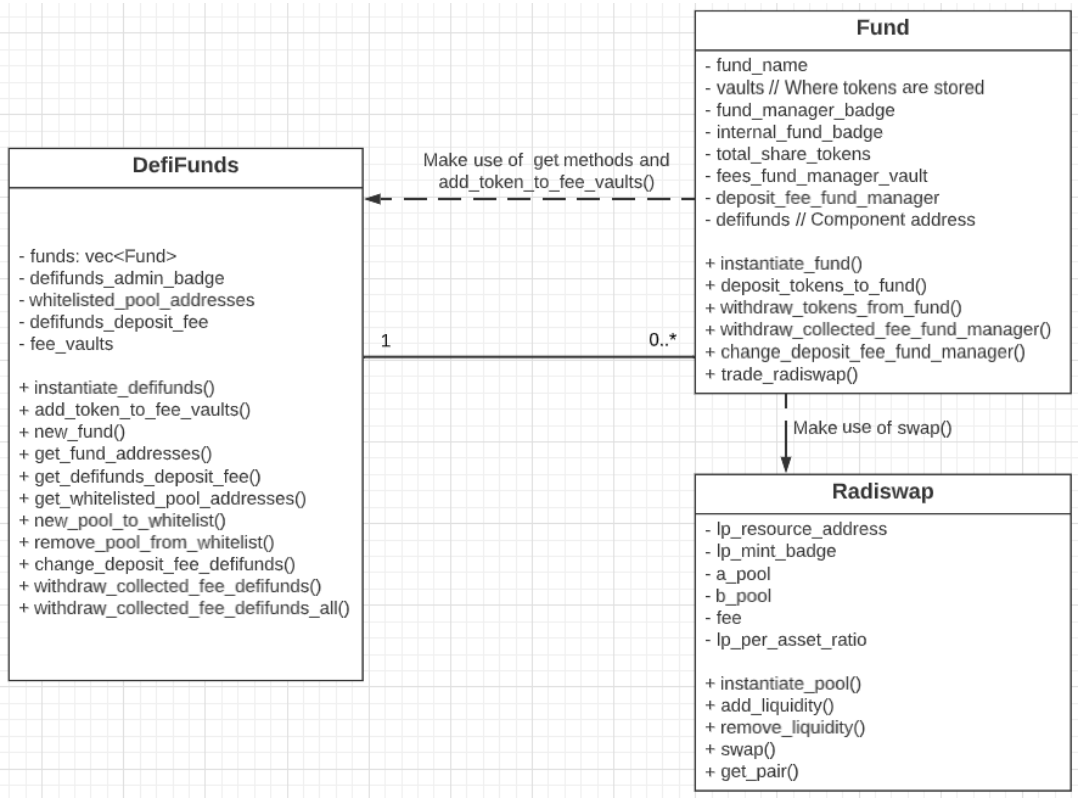

Figure 4.1: UML digram showing the relation between the blueprints.

noe

4.2.1 Fund - Blueprint

The main purpose of this blueprint is to create a single fund where normal users can deposit and withdraw tokens. The fund manager can trade with the assets in the fund, but not be able to withdraw the assets to his own account. The Fund blueprint also uses a whitelist from the DefiFunds blueprint that decides what trading pools the fund manager can trade with. This is to make sure that the fund manager doesn't trade with malicious trading pools that could result in impermanent loss for the users. The Fund blueprint also gives the fund manager a fee based on the value deposited to the fund. Deposit fee was chosen to follow the last requirement in Section 4.1. Down below is a list of the main methods in the Fund blueprint. You can see the implementation of the methods in Section A.1

deposit_tokens_to_fund Allows normal users to deposit tokens to the fund. When you use this method you need to deposit tokens in almost the same ratio as the fund. If there exist 20 BTC and 10 ETH in the fund you can for example deposit 2 BTC and 1 ETH. In return, you will get tokens representing your share of the total pool. This method will most likely be combined with another method in the future that swaps a single token type for the different token types needed for this method. From a user interface perspective, it would make more sense to send one token type instead of many different ones.
Line 108-151, A.1

witdraw_token_from_fund Allows normal users to send share tokens back to the fund component and receive tokens equal to the share they deposited. They will receive a share of all the different tokens in the fund. This method will likely be combined with another method in the future that converts all the different token types to a single token type.
Line 155-174, A.1

withdraw_collected_fee_fund_manager Allows the fund manager to withdraw share tokens from the vault, which contains the collected fee.
Line 183-186, A.1

change_deposit_fee_fund_manager Allows the fund manager to change the fee that goes to the fund manager vault when a user deposits tokens to the fund.
Line 188-193, A.1

trade_radiswap Allows the fund manager to trade with all tokens in the fund. This method makes use of the swap method from the Radiswap blueprint as of now, but will likely be changed to make use of a swap method from another blueprint in the future. The fund manager can only trade using pools defined in the whitelist.
Line 197-217, A.1

4.2.2 DefiFunds - Blueprint

The main purpose of the DefiFunds blueprint is to keep track of the different funds and control the whitelist that the different funds make use of. There is introduced a time delay of 7 days on all new pools added to the whitelist. This gives users time to withdraw their tokens in case the admin acts dishonestly. If the admin account gets compromised and malicious pools are added, it will give the admin time to call the "remove_pool_from_whitelist" method and warn users to withdraw their tokens. How well this corresponds with the safety of the user funds requirement, is discussed in the last example in Section 4.3.3. The DefiFunds blueprint also collects a fee to the owner of DefiFunds. Here is a list of the main functions and methods in the DefiFunds blueprint. You can see the implementation of the functions and methods in Section A.2.

instantiate_defifunds This function creates the DefiFunds component. The caller of this function will receive a badge that is used to control the functions that only the DefiFunds admin has access to.
Line 17-45, A.2

new_fund Allows people to create a new fund. The creator of this fund will receive a fund manager badge that is used to control the methods that only the fund manager has access to. It makes use of the "instantiate_fund" function in A.1 line 22-80.
Line 65-76, A.2

new_pool_to_whitelist Allows the DefiFunds admin to add a trading pool to the whitelist. The trading pool will only be valid after 300 epochs (about seven days).
Line 96-98, A.2

remove_pool_from_whitelist Allows the DefiFunds admin to remove a trading pool from the whitelist.
Line 100-102, A.2

change_deposit_fee_defifunds Allows the DefiFunds admin to change the fee that goes to the DefiFunds admin vault when a user deposit tokens to a fund.
Line 104-108, A.2

withdraw_collected_fee_defifunds_all Allows the DefiFunds admin to withdraw the collected fee he has gotten from users depositing to funds.
Line 111-117, A.2

4.3 Demo

4.3.1 Getting started

In this section, I will explain step by step how to test the blueprints in the Radix Engine simulator. I will highly recommend doing it in your own local simulator to get the most out of this section. If you haven't installed essentials for Scrypto yet take a look at the Radix docs first [39]. After you successfully have installed the Scrypto toolchain you can clone the GitHub repo "https://github.com/tobben1998/scrypto.git", switch to the branch named "SpecializationProject" and move into the "defi_fund" folder. Make sure to use scrypto version 0.6.0. You can then follow the steps below to get started. The commands should be formatted correctly to not encounter formatting errors when copying and pasting. If you get problems you can copy from the readme file instead.

Start by resetting the Radix Engine simulator.

```
resim reset
```

You will then need to create some new accounts.

```
op1=$(resim new-account)
export pk1=$(echo "$op1" | sed -nr "s/Private key: ([[:alnum:]_]+)/\1/p")
export acc1=$(echo "$op1" | sed -nr "s/Account component address: ([[:alnum:]_]+)/\1/p")
op2=$(resim new-account)
export pk2=$(echo "$op2" | sed -nr "s/Private key: ([[:alnum:]_]+)/\1/p")
export acc2=$(echo "$op2" | sed -nr "s/Account component address: ([[:alnum:]_]+)/\1/p")
op3=$(resim new-account)
export pk3=$(echo "$op3" | sed -nr "s/Private key: ([[:alnum:]_]+)/\1/p")
export acc3=$(echo "$op3" | sed -nr "s/Account component address: ([[:alnum:]_]+)/\1/p")
op4=$(resim new-account)
export pk4=$(echo "$op4" | sed -nr "s/Private key: ([[:alnum:]_]+)/\1/p")
export acc4=$(echo "$op4" | sed -nr "s/Account component address: ([[:alnum:]_]+)/\1/p")
```

Create some new tokens, and send tokens to the different accounts you just created.

```
resim set-default-account $acc1 $pk1

export xrd=resource_sim1qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqzqu57yag
op5=$(resim new-token-fixed --name Bitcoin --symbol BTC 10000000)
export btc=$(echo "$op5" | sed -nr "s/.*Resource: ([[:alnum:]_]+)/\1/p")
op6=$(resim new-token-fixed --name Ethereum --symbol ETH 10000000)
export eth=$(echo "$op6" | sed -nr "s/.*Resource: ([[:alnum:]_]+)/\1/p")
op7=$(resim new-token-fixed --name Tether --symbol USDT 10000000)
export usdt=$(echo "$op7" | sed -nr "s/.*Resource: ([[:alnum:]_]+)/\1/p")
op8=$(resim new-token-fixed --name Dogecoin --symbol DOGE 10000000)
export doge=$(echo "$op8" | sed -nr "s/.*Resource: ([[:alnum:]_]+)/\1/p")

resim transfer 100000 $btc $acc2
resim transfer 100000 $btc $acc3
resim transfer 100000 $btc $acc4
resim transfer 100000 $eth $acc2
resim transfer 100000 $eth $acc3
resim transfer 100000 $eth $acc4
resim transfer 100000 $usdt $acc2
resim transfer 100000 $usdt $acc3
resim transfer 100000 $usdt $acc4
resim transfer 100000 $doge $acc2
resim transfer 100000 $doge $acc3
resim transfer 100000 $doge $acc4
```

Publish the package containing the three blueprints.

```
pkg=$(resim publish ".")
export pkg=$(echo "$pkg" | sed -nr "s/Success! New Package: ([[:alnum:]_]+)/\1/p")
```

Create some new trading pools using the Radiswap blueprint.

```
pools=$(resim run "./transactions/instantiate_radiswap_pools_acc1.rtm")
export pool_btc_usdt=$(echo "$pools" | sed -nr "s/.*Component: ([[:alnum:]_]+)/\1/p" | sed '1q;d')
export pool_eth_usdt=$(echo "$pools" | sed -nr "s/.*Component: ([[:alnum:]_]+)/\1/p" | sed '2q;d')
export pool_doge_usdt=$(echo "$pools" | sed -nr "s/.*Component: ([[:alnum:]_]+)/\1/p" | sed '3q;d')
```

Instantiate DefiFunds using the DefiFunds blueprint.

```
defifunds=$(resim run "./transactions/instantiate_defifunds_acc1.rtm")
export defifunds_admin_badge=$(echo "$defifunds" | sed -nr "s/.*Resource: ([[:alnum:]_]+)/\1/p")
export defifunds=$(echo "$defifunds" | sed -nr "s/.*Component: ([[:alnum:]_]+)/\1/p")
```

Finally, add some Radiswap pools to the whitelist in the DefiFunds component. Because of the 300 epoch delay feature explained in 4.2.2, you also need to move 300 epochs forward in time.

```
resim set-current-epoch 0
resim call-method $defifunds new_pool_to_whitelist $pool_btc_usdt --proofs 1,$defifunds_admin_badge
resim call-method $defifunds new_pool_to_whitelist $pool_eth_usdt --proofs 1,$defifunds_admin_badge
resim call-method $defifunds new_pool_to_whitelist $pool_doge_usdt --proofs 1,$defifunds_admin_badge
resim set-current-epoch 300
```

You have now created all the essential components in the Dapp, and are ready to start going through some examples.

4.3.2 Example - Basic features

In this example, I will go through the basic features of the DefiFunds Dapp. Creating a fund, depositing and withdrawing from a fund, trading with the tokens in a fund, and withdrawing the collected fee.

Start by creating a new fund using account 2, and change the deposit fee. Let the name be "DegenFund", place a bucket with 100 USDT, and set initial share tokens to be 100. Account 2 will now be the fund manager of "DegenFund". He will receive a badge used to get access to the methods only available for the fund manager, and some share tokens representing his share of the fund.

```
resim set-default-account $acc2 $pk2

fund=$(resim call-method $defifunds new_fund "DegenFund" 100,$usdt 100)
export fund_manager_badge=$(echo "$fund" | sed -nr "s/.*Resource: ([[:alnum:]_]+)/\1/p" | sed '1q;d')
export share_token=$(echo "$fund" | sed -nr "s/.*Resource: ([[:alnum:]_]+)/\1/p" | sed '3q;d')
export fund=$(echo "$fund" | sed -nr "s/.*Component: ([[:alnum:]_]+)/\1/p")

resim call-method $fund change_deposit_fee_fund_manager 1 --proofs 1,$fund_manager_badge
```

The fund you created now contains 100 USDT. Let's swap 20 of those USDT for DOGE.

```
resim set-default-account $acc2 $pk2
resim call-method $fund trade_radiswap $usdt 20 $pool_doge_usdt --proofs 1,$fund_manager_badge
```

The fund now contains 80 USDT and 199.99 DOGE. You can verify by calling:

```
resim show $fund
```

Switch to another user for example account 3, and deposit 40 USDT and 100 DOGE. The command below uses a transaction manifest, and you can edit the parameters by editing the ".rtm" file. See A.4 for code details. The method you call needs tokens in about the same ratio as the fund. If you don't have the tokens needed, you can use the Radiswap component and swap to the correct amounts. Swapping and depositing can be done in the same transaction using a transaction manifest.

```
resim set-default-account $acc3 $pk3
resim run transactions/deposit_usdt_and_doge_acc3.rtm
```

You have now deposited to the fund and received 49 share tokens. One share token has been taken as a fee. You can verify by doing:

```
resim show $acc3
resim show $fund
resim show $defifunds
```

Let's switch back to the fund manager account and do some trades with the tokens in the fund.

```
resim set-default-account $acc2 $pk2
resim call-method $fund trade_radiswap $usdt 40 $pool_btc_usdt --proofs 1,$fund_manager_badge
resim call-method $fund trade_radiswap $usdt 40 $pool_eth_usdt --proofs 1,$fund_manager_badge
resim call-method $fund trade_radiswap $doge 100 $pool_doge_usdt --proofs 1,$fund_manager_badge
resim show $fund
```

Account 3 holds 49 share tokens, and there exists a total of 150 share tokens. When he calls the withdraw method, he will get almost 1/3 of the tokens in the fund. If you want to get one token instead of many different ones when you withdraw, you can use the Radiswap component to swap them into one. You can use the transaction manifest to do it all in a single transaction.

```
resim set-default-account $acc3 $pk3
resim call-method $fund withdraw_tokens_from_fund 49,$share_token
resim show $fund
resim show $acc3
```

Some fees have been collected from users depositing to the fund. You can collect the fees by calling the methods below:

```
resim set-default-account $acc1 $pk1
resim call-method $defifunds withdraw_collected_fee_defifunds_all --proofs 1,$defifunds_admin_badge
resim show $acc1

resim set-default-account $acc2 $pk2
resim call-method $fund withdraw_collected_fee_fund_manager --proofs 1,$fund_manager_badge
resim show $acc2
```

You have now gone through a simple example of how DefiFunds work. If you want to explore more, you can for example create multiple funds. To get a better understanding of how the components and methods work, you should check the source files. See Appendix A

4.3.3 Examples - Misusing methods

1. Withdraw from the fund

The first example is just to give a clearer explanation of how components work. The only method that lets you withdraw tokens from the fund directly is the "witdraw_tokens_from_fund" method. There is no owner of the fund component, and all rules for who can do what with the component, are defined in the blueprint. You can for example try to call the "withdraw_tokens_from_fund" method with more share tokens than you have.

```
resim set-default-account $acc2 $pk2
resim call-method $fund withdraw_tokens_from_fund 120,$share_token
```

The method will obviously fail because you don't have enough share tokens. There is no other way to directly take tokens from the fund to your account. The other way tokens are moved from the fund is through the "trade_radiswap" method. Potential misuse of this method is covered in the last example.

2. Withdraw collected fee

Let's test to call the "withdraw_collected_fee_fund_manager" method with another user than the fund manager.

```
resim set-default-account $acc1 $pk1
resim call-method $fund withdraw_collected_fee_fund_manager
```

If you do so, you will get an authorization error. It fails because of the access rule defined in the "instantiate_fund" function. See A.1 line 54. The only one that can call the method is the account holding the fund manager badge.

3. Adding a malicious pool to the whitelist

Let's first explain how a trading pool can be malicious and then go into two scenarios. When doing a trade between token A and token B you are supposed to get the same dollar amount of token B as you swap with token A minus some fees. A pool can be malicious if that is not the case. A person can create a pool with 100 tokens he created him selves and pool them with 1 USDT. If a trader decides to use that pool, he would basically give the creator of that pool USDT and get a token worth nothing in return.

A scenario where this can be misused is if a fund manager is able to compromise the DefiFunds admin wallet. He will then add a malicious pool to the whitelist and try to trade with it.

```
resim set-default-account $acc1 $pk1
pools=$(resim run "./transactions/instantiate_malicious_pool_acc1.rtm")
export pool_malicious=$(echo "$pools" | sed -nr "s/.*Component: ([[:alnum:]_]+)/\1/p" | sed '1q;d')
resim call-method $defifunds new_pool_to_whitelist $pool_malicious --proofs 1,$defifunds_admin_badge

resim set-default-account $acc2 $pk2
resim call-method $fund trade_radiswap $btc 20 $pool_malicious --proofs 1,$fund_manager_badge
```

He won't have access to trade on that pool before the 300 epochs have occurred. The creator of DefiFunds can then call the "remove pool_from_whitelist" method and warn users to withdraw their funds. As long as the owner still has access to his wallet, like he most likely has, he can continue to call the "remove_pool_from_whitelist" method if the pool is re-added. The users will then get in practice indefinitely time to withdraw their funds.

Another scenario that could occur, is that the admin of DefiFunds decides to add a malicious pool himself, because he controls a large fund and wants to empty it. Some users will likely monitor the whitelist and warn other users to withdraw. To expect that all users would be able to withdraw their tokens, from the fund that the admin of DefiFunds control, within seven days is optimistic. However, a huge portion will likely be able to do so. To further increase the safety of this unlikely event, as incentives most likely are going against the admin to act dishonestly, could be to extend the time delay.

To totally avoid the trust for an admin to control a whitelist with time delays, a whitelist can be added when the DefiFunds component is created. The inconvenience here is that a new DefiFunds component would need to be made if new trading pools should be added. Another complex solution that would minimize the chance of malicious pools being added, could be to create a decentralized autonomous organization that controls the whitelist.

Discussion

5.1 Design and implementation

A general idea in developing is the desire to build modules as independently as possible, but both DefiFunds and Funds make use of each other in Figure 4.1. It is evident that DefiFunds need to use the Fund blueprint since DefiFunds have a list of all the funds, but Fund also makes use of DefiFunds. The main reason for having it that way is that Fund uses the whitelist. When I first designed the blueprint I put the whitelist in the Fund component, but figured out that it was better to store the whitelist in a single place, instead of a whitelist for every fund. When only having one whitelist it will be clearer for the users when new pools are added to the whitelist. The vault with admin fees was also first stored in the Fund blueprint. It was later moved to the DefiFunds blueprint, to keep all admin methods in the DefiFund blueprint, and all the fund manager methods in the Fund blueprint.

5.2 Weakness with the whitelist

The one potential misuse of this Dapp that I am aware of, is that malicious pools are added to the whitelist. A malicious pool is explained in the last example in Section 4.3.3. The first possible scenario is that the admin wallet gets compromised and a malicious pool is added. It is unlikely to cause any trouble, because of the time delay of seven days, as the admin just can call the "remove_from_whitelist_method". In my opinion, the bigger threat is if the admin himself decides to add a malicious pool. Most of the users will likely be warned by other users and withdraw their funds within seven days. The only funds that will be affected by malicious pools being added, are the funds that the DefiFunds admin control or his partner(s). It wouldn't make sense for other fund managers to trade with malicious pools that the admin has added. The question to ask here, is if the incentives for the admin to add malicious pools are greater than not doing it. He would likely earn some money in the short term, but will lose his income stream from the fees collected. The solution may be good enough, but it obviously can be improved, so I may need to think further on possible better solutions.

5.3 Developing on an early phase network

Developing on an early-phase network has caused some challenges. The language in itself has been relatively easy to understand, but the documentation and tutorials have been somewhat limited. I haven't used the Scrypto 101 course, mentioned in Section 3.2, for learning the language as it came quite late in my developing period. During my learning period, I have mainly used other GitHub repositories to understand how the language worked, and asked questions in Radix's discord channel. Several of the good GitHub repositories have been made with earlier scrypto versions which also was a challenge.

Another challenge with the Dapp I have made is that it depends on smart contracts to Dapps that are not made yet. Had I chosen to develop on Ethereum instead, production-ready smart contracts for DEXes have already been made. The "trade_radiswap" method will for example need to be changed. "trade_radiswap" make use of a swap method from the Radiswap blueprint, but will need use a swap method from a production-ready blueprint. The Radiswap blueprint also lacked methods that I would need to make the method mentioned in: "deposit_tokens_to_fund" Section 4.2.1. I would need a method that gives the price of a token. I would also need a method that gives the estimated token output based on token input. It wouldn't be a huge problem to make those methods my selves, but it would have been better to have already-made methods that I know for sure will be in a final DEX blueprint.

The decision to develop on the Radix network instead of the Ethereum network has had its own challenges, but has in my opinion been very limited compared to the process of developing on Ethereum. The Scrypto language has through my experience proved to be rather intuitive to learn. The bigger question to ask is if the Radix network will get mass adoption, and if there is any point in making a Dapp on this network at all.

Conclusion

6.1 Conclusion

A proof of concept Dapp called DefiFunds has been made on the Radix network. The Dapp allows traders to create decentralized funds and lets everyday people invest in them. Four of five requirements specified in 4.1 are met if no bugs exists that the developer of this project is unaware of. The second requirement, "No one should have access to get tokens from the fund directly or indirectly apart from when users withdraw their share of the fund," is not met. It can be met if the decision to hardcode the whitelist instead of letting an admin control the whitelist is taken. The downside is that flexibility is reduced, and more about this discussion can be read in the last example in 4.3.3. The requirement may need to be adjusted to not be an absolute if flexibility is wanted.

The work on this proof of concept Dapp will serve as a starting point for a fully functional Dapp. There is still a lot of work to be done, especially on the frontend. Some of the work is not possible yet, because the Dapp depends on other applications to build their blueprints first. The Radix network is in its early phase, and the success of this Dapp is heavily dependent on Radix getting mass adoption.

6.2 Future work

There exists several opportunities for developing further on this proof of concept Dapp. I have listed essential tasks that needs to be done and some possible improvements. I recommend waiting for betanet to go live before parts on frontend, related to integration with the Radix network are started on.

Essential tasks

- Change the "trade_radiswap" method to use the correct swapping method. Need to wait for the release of the blueprint that Ociswap (current biggest DEX on Radix) is going to use.
- Make a method that lets users deposit a single token to the fund. An idea could be to make a method that takes in one token, and returns a vector with tokens in the same ratio as the fund. This method can then be combined with the already-made deposit method.
- Make a method that swaps all tokens to a single token. This method will be combined with the withdraw method. A user will then be able to only receive one token type when he withdraws from the fund.
- Deploy the Dapp to betanet, and finally to mainnet when it goes live.
- Make a frontend that lets users call the methods.

Possible tasks

- Make methods that make it easier to get a good frontend. For example, methods that will be used to show statistics for the fund, or methods that will let the fund manager have a way to communicate how he is planning to trade in the fund. Find out if this is best done with Scrypto or if other better solutions exists.
- Make a good-looking user interface.
- Possible to substitute share tokens, with NFTs representing the number of share tokens, and the value of the share token at the deposit time. Fees that only will be taken if the fund manager trade with profits can then be implemented. For example, take a withdraw fee of 10% of the profits made relative to when the user deposited and withdrew his tokens.
- Find out if there exists a safer solution to control the whitelist of pools. For example, let a decentralized autonomous organization control the whitelist.
- Get the Dapp audited.

Bibliography

[1] Ethereum TxHash: 0x01afae47b0c98731b5d20c776e58bd8ce5c2c89ed4bd3f8727fad3ebf32e9481. URL: https://etherscan.io/tx/0x01afae47b0c98731b5d20c776e58bd8ce5c2c89ed4bd3f8727fad3ebf32e9481.

[2] Ethereum TxHash: 0x13bfe9bea4e21ea532988350a8b7eef002686b3b655f19f8760210d1069ab585. URL: https://etherscan.io/tx/0x13bfe9bea4e21ea532988350a8b7eef002686b3b655f19f8760210d1069ab585.

[3] 482.solutions. *Distributed Ledger Technology and it's types. |Medium*. Feb. 2018. URL: https://medium.com/482-labs/distributed-ledger-technology-and-its-types-ad76565ae76 (visited on 10th Sept. 2022).

[4] *5.2 Ledger State |IOTA*. URL: https://wiki.iota.org/IOTA-2.0-Research-Specifications/5.2LedgerState (visited on 12th Sept. 2022).

[5] Aave. *aave-protocol/Aave_Protocol_Whitepaper_v1_0 |Aave*. Jan. 2020. URL: https://github.com/aave/aave-protocol/blob/master/docs/Aave_Protocol_Whitepaper_v1_0.pdf.

[6] Eray Arda Akartuna et al. *NFTs and Financial Crime |Elliptic.co*. 2022. URL: https://www.elliptic.co/resources/nfts-financial-crime.

[7] *Bitcoin Energy Consumption Index - Digiconomist*. URL: https://digiconomist.net/bitcoin-energy-consumption (visited on 1st Oct. 2022).

[8] Lorenz Breidenbach et al. *An In-Depth Look at the Parity Multisig Bug*. July 2017. URL: https://hackingdistributed.com/2017/07/22/deep-dive-parity-bug/ (visited on 2nd Oct. 2022).

[9] Vitalik Buterin. *Why sharding is great: demystifying the technical properties*. Apr. 2021. URL: https://vitalik.ca/general/2021/04/07/sharding.html.

[10] Florian Cäsar et al. 'A Parallelized BFT Consensus Protocol for Radix'. In: (2020). URL: https://assets.website-files.com/6053f7fca5bf627283b582c2/608811e3f5d21f235392fee1_Cerberus-Whitepaper-v1.01.pdf.

[11] Chainlink. *What Is the Blockchain Oracle Problem? Why Can't Blockchains Solve It?* Aug. 2020. URL: https://blog.chain.link/what-is-the-blockchain-oracle-problem/ (visited on 5th Oct. 2022).

[12] Emanuel Coen. *Learn about Token Allowances & How to keep your Ethereum Secure*. 2020. URL: https://cryptotesters.com/blog/token-allowances (visited on 3rd Oct. 2022).

[13] *Consensus mechanisms |Ethereum.org*. Sept. 2022. URL: https://ethereum.org/en/developers/docs/consensus-mechanisms/ (visited on 20th Sept. 2022).

[14] Brady Dale. *Ethereum 2.0's Composability Concerns, Explained - CoinDesk*. Oct. 2020. URL: https://www.coindesk.com/tech/2020/10/13/will-a-sharded-ethereum-be-flexible-enough-for-decentralized-finance/ (visited on 30th Nov. 2022).

[15] *Decentralized Data Model |ChainlinkDocumentation*. URL: https://docs.chain.link/architecture-overview/architecture-decentralized-model (visited on 15th Oct. 2022).

[16] *dHEDGE*. URL: https://www.dhedge.org/ (visited on 1st Dec. 2022).

[17] etherscan.io. *Average Daily Transaction Fee*. URL: https://etherscan.io/chart/avg-txfee-usd (visited on 7th Oct. 2022).

[18] etherscan.io. *Ethereum Daily Transactions Chart*. URL: https://etherscan.io/chart/tx (visited on 7th Oct. 2022).

[19] *FAQ: Blockchain & DeFi Basics |RadixDLT*. Sept. 2022. URL: https://learn.radixdlt.com/article/whats-the-difference-between-proof-of-stake-pos-and-delegated-proof-of-stake-dpos (visited on 12th Oct. 2022).

[20] The RADIX FOUNDATION. *Radix DeFi White paper*. Aug. 2020. URL: https://www.radixdlt.com/post/defi-whitepaper-how-radix-is-building-the-future-of-defi (visited on 10th Oct. 2022).

[21] Fredrik Haga. *DEX Tracker - Decentralized Exchanges Trading Volume |defiprime.com*. URL: https://defiprime.com/dex-volume (visited on 3rd Oct. 2022).

[22] Jelle Hellings et al. *Cerberus: Minimalistic Multi-shard Byzantine-resilient Transaction Processing*. 2020. DOI: 10.48550/ARXIV.2008.04450. URL: https://arxiv.org/abs/2008.04450.

[23] *How The Merge impacted ETH supply |Ethereum.org*. Sept. 2022. URL: https://ethereum.org/en/upgrades/merge/issuance/ (visited on 30th Sept. 2022).

[24] Markus Jakobsson and Ari Juels. 'Proofs of Work and Bread Pudding Protocols(Extended Abstract)'. In: *Secure Information Networks*. Springer US, 1999, pp. 258–272. DOI: 10.1007/978-0-387-35568-9_18.

[25] Yuen Lo and Francesca Medda. 'Uniswap and the rise of the decentralized exchange'. In: (2020). URL: https://mpra.ub.uni-muenchen.de/103925/.

[26] Loi Luu et al. 'Making Smart Contracts Smarter'. In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, Oct. 2016. DOI: 10.1145/2976749.2978309.

[27] Jake Mai. *Scrypto 101 BETA |radixacademy*. Nov. 2022. URL: https://radixacademy.mylearnworlds.com/course/scrypto-101 (visited on 29th Nov. 2022).

[28] Rachel McIntosh. *Is the Binance Smart Chain Centralized? Messari Researchers Raise Concerns*. Apr. 2021. URL: https://www.financemagnates.com/cryptocurrency/news/is-the-binance-smart-chain-centralized-messari-researchers-raise-concerns/ (visited on 7th Oct. 2022).

[29] Kirsty Moreland. *Crypto's Greatest Weakness? Blind Signing, Explained |Ledger.com*. 2022. URL: https://www.ledger.com/academy/cryptos-greatest-weakness-blind-signing-explained (visited on 3rd Oct. 2022).

[30] Kirsty Moreland. *The Safest Way to Use MetaMask With Ledger Hardware Wallet |Ledger*. 2022. URL: https://www.ledger.com/academy/security/the-safest-way-to-use-metamask (visited on 7th Oct. 2022).

[31] Satoshi Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System*. Oct. 2008. URL: https://bitcoin.org/bitcoin.pdf.

[32] Jason Scharfman. 'Decentralized Finance (DeFi) Compliance and Operations'. In: *Cryptocurrency Compliance and Operations*. Springer International Publishing, Nov. 2021, pp. 171–186. URL: https://link.springer.com/chapter/10.1007/978-3-030-88000-2_9.

[33] Corwin Smith. *Proof-of-stake (PoS) |Ethereum.org*. Sept. 2022. URL: https://ethereum.org/en/developers/docs/consensus-mechanisms/pos/ (visited on 22nd Sept. 2022).

[34] Ali Sunyaev. *Internet computing: Principles of Distributed systems and emerging internet-based technologies*. Springer Nature, 2020. Chap. 9. URL: https://link.springer.com/content/pdf/10.1007/978-3-030-34957-8.pdf.

[35] *Tokens - OpenZeppelin Docs*. URL: https://docs.openzeppelin.com/contracts/4.x/tokens (visited on 28th Nov. 2022).

[36] Nikhil Vadgama, Jiahua Xu and Paolo Tasca. *Enabling the Internet of Value*. URL: https://link.springer.com/content/pdf/10.1007/978-3-030-78184-2.pdf.

[37] RDX Works. *Cerberus Infographic Series - Chapter I |TheRadixBlog|RadixDLT*. URL: https://www.radixdlt.com/post/cerberus-infographic-series-chapter-i (visited on 30th Nov. 2022).

[38] RDX Works. *FAQ: Radix Overview*. URL: https://learn.radixdlt.com/article/what-is-the-radix-roadmap (visited on 7th Dec. 2022).

[39] RDX Works. *Install the Scrypto Toolchain :: Radix Documentation*. URL: https://docs.radixdlt.com/main/scrypto/getting-started/install-scrypto.html (visited on 12th Dec. 2022).

[40] RDX Works. *Radiswap*. Sept. 2022. URL: https://github.com/radixdlt/scrypto-examples/tree/main/defi/radiswap (visited on 1st Nov. 2022).

[41] RDX Works. *Resources*. URL: https://docs.radixdlt.com/main/scrypto/scrypto-lang/resources.html (visited on 5th Dec. 2022).

[42] RDX Works. *Setting Access Rules*. URL: https://docs.radixdlt.com/main/scrypto/scrypto-lang/access-control/access-setting.html (visited on 5th Dec. 2022).

[43] RDX Works. *Transaction Manifest*. URL: https://docs.radixdlt.com/main/scrypto/transaction-manifest/intro.html (visited on 3rd Nov. 2022).

[44] Zibin Zheng et al. 'An overview on smart contracts: Challenges, advances and platforms'. In: *Future Generation Computer Systems* 105 (Apr. 2020), pp. 475–491. DOI: 10.1016/j.future.2019.12.019.

Appendix

If you prefer to have the code files you can find them at:
https://github.com/tobben1998/scrypto/tree/SpecializationProject/defi_fund

A.1 Fund - Blueprint

```
use scrypto::prelude::*;
use crate::radiswap::*;
use crate::defifunds::*;

blueprint! {


    struct Fund {
        fund_name: String,
        vaults: HashMap<ResourceAddress, Vault>, //where all the tokens in the fund are stored
        fund_manager_badge: ResourceAddress,
        internal_fund_badge: Vault,
        total_share_tokens: Decimal,
        fees_fund_manager_vault: Vault,
        deposit_fee_fund_manager: Decimal,
        defifunds: ComponentAddress, //defifunds ComponentAddress to get access to whitelist and deposit fee

    }

    impl Fund {

        pub fn instantiate_fund(
            fund_name: String,
            token: Bucket,
            initial_supply_share_tokens: Decimal,
            defifunds: ComponentAddress
        ) -> (ComponentAddress, Bucket, Bucket) {

            let fund_manager_badge: Bucket = ResourceBuilder::new_fungible()
                .divisibility(DIVISIBILITY_NONE)
                .metadata("name", "Fund manager badge")
                .metadata("desciption", "Badge used for managing the fund.")
                .initial_supply(1);


            let internal_fund_badge: Bucket = ResourceBuilder::new_fungible()
                .divisibility(DIVISIBILITY_NONE)
                .metadata("name", "Internal fund badge")
                .metadata("desciption", "Badge that has the auhority to mint and burn share tokens.")
                .initial_supply(1);


            let share_tokens: Bucket = ResourceBuilder::new_fungible()
                .divisibility(DIVISIBILITY_MAXIMUM)
                .metadata("name", "hare tokens")
                .metadata("description", "Tokens used to show what share of the fund you have")
                .mintable(rule!(require(internal_fund_badge.resource_address())),LOCKED)
                .burnable(rule!(require(internal_fund_badge.resource_address())),LOCKED)
                .initial_supply(initial_supply_share_tokens);


            let access_rules = AccessRules::new()
                .method("change_deposit_fee_fund_manager", rule!(require(fund_manager_badge.resource_address())))
                .method("withdraw_collected_fee_fund_manager",
↪   rule!(require(fund_manager_badge.resource_address())))
```

```
55                  .method("trade_radiswap", rule!(require(fund_manager_badge.resource_address())))
56                  .default(rule!(allow_all));
57
58
59
60              let mut vaults = HashMap::new();
61              vaults.insert(token.resource_address(),Vault::new(token.resource_address())); // adding a new vault.
62              vaults.get_mut(&token.resource_address()).unwrap().put(token); //putting tokens in the vault
63
64
65              let mut component = Self {
66                  fund_name: fund_name,
67                  fund_manager_badge: fund_manager_badge.resource_address(),
68                  internal_fund_badge: Vault::with_bucket(internal_fund_badge),
69                  vaults: vaults,
70                  total_share_tokens: initial_supply_share_tokens,
71                  fees_fund_manager_vault: Vault::new(share_tokens.resource_address()),
72                  deposit_fee_fund_manager: dec!(0),
73                  defifunds: defifunds
74              }
75              .instantiate();
76              component.add_access_check(access_rules);
77
78              (component.globalize(),fund_manager_badge, share_tokens)
79
80          }
81
82
83          //////////////////
84          //helper method///
85          //////////////////
86
87          fn add_token_to_fund(&mut self, token: Bucket){
88              let resource_address=token.resource_address();
89
90              //create a new vault if not exsisting
91              if !self.vaults.contains_key(&resource_address){
92                  let key=resource_address;
93                  let value=Vault::new(resource_address);
94                  self.vaults.insert(key,value);
95              }
96              //put token in the vault with specified resource address.
97              self.vaults.get_mut(&resource_address).unwrap().put(token);
98          }
99
100
101
102
103         ////////////////////////
104         //methods for everyone//
105         ////////////////////////
106
107
108         //method for depositing tokens to the fund. You need to deposit each token that exists in the pool.
109         //tokens will be taken in the same ratio as the pool has, the rest of the tokens will be returned back.
110         pub fn deposit_tokens_to_fund(&mut self, mut tokens: Vec<Bucket>) -> (Bucket, Vec<Bucket>) {
111
112             //calculate min_ratio to find out how much you should take from each bucket,
113             //so there is enough to take, an the ratio in the pool remains the same. The rest will be given back
114             let mut ratio=tokens[0].amount()/self.vaults.get_mut(&tokens[0].resource_address()).unwrap().amount();
115             let mut min_ratio=ratio;
116             for token in &tokens{
117                 ratio=token.amount()/(self.vaults.get_mut(&token.resource_address()).unwrap().amount());
118                 if ratio<min_ratio{
119                     min_ratio=ratio;
120                 }
121             }
122
123             //take from buckets, and put them into the fund.
124             for token in tokens.iter_mut(){
125                 let amount=min_ratio*(self.vaults.get_mut(&token.resource_address()).unwrap().amount());
126                 self.add_token_to_fund(token.take(amount));
127             }
128
129             //mint new sharetokens
130             let new_share_tokens=min_ratio*self.total_share_tokens;
131             self.total_share_tokens += new_share_tokens;
132             let resource_manager = borrow_resource_manager!(self.fees_fund_manager_vault.resource_address());
133             let mut share_tokens = self
134                 .internal_fund_badge
135                 .authorize(|| resource_manager.mint(new_share_tokens));
136
137             //deposit fee to the fund manager and to defifunds
138             let defifunds: DefifundsComponent=self.defifunds.into();
139
140             let fee_fund_manager=(self.deposit_fee_fund_manager/dec!(100))*share_tokens.amount();
```

```
            let fee_defifunds=(defifunds.get_defifunds_deposit_fee()/dec!(100))*share_tokens.amount();

            self.fees_fund_manager_vault.put(share_tokens.take(fee_fund_manager));
            defifunds.add_token_to_fee_vaults(share_tokens.take(fee_defifunds));

            info!("Returned share tokens: {:?}.", share_tokens.amount());
            info!("share tokens fee: {:?}.", fee_fund_manager+fee_defifunds);


            (share_tokens, tokens)
        }



        //method that withdraw tokens from the fund relative to how much sharetokens you put into the method.
        pub fn withdraw_tokens_from_fund(&mut self, share_tokens: Bucket) -> Vec<Bucket> {
            assert!(share_tokens.resource_address()==self.fees_fund_manager_vault.resource_address(),"Wrong tokens sent. You need to send share tokens.");

            //take fund from vaults and put into a Vec<Bucket> called tokens
            let mut tokens = Vec::new();
            let your_share = share_tokens.amount()/self.total_share_tokens;
            for vault in self.vaults.values_mut(){
                info!("Withdrew {:?} {:?}.", your_share*vault.amount(), vault.resource_address());
                tokens.push(vault.take(your_share*vault.amount()));
            }

            //burn sharetokens
            self.total_share_tokens -= share_tokens.amount();
            let resource_manager = borrow_resource_manager!(self.fees_fund_manager_vault.resource_address());
            self.internal_fund_badge.authorize(|| resource_manager.burn(share_tokens));


            tokens
        }



        ////////////////////////////
        //methods for fund manager//
        ////////////////////////////


        pub fn withdraw_collected_fee_fund_manager(&mut self) -> Bucket{
            info!("Withdrew {:?} sharetokens from vault.", self.fees_fund_manager_vault.amount());
            self.fees_fund_manager_vault.take_all()
        }

        pub fn change_deposit_fee_fund_manager(&mut self, new_fee: Decimal){
            assert!(new_fee >= dec!(0) && new_fee <= dec!(5),"Fee need to be in range of 0% to 5%.");
            self.deposit_fee_fund_manager=new_fee;
            info!("Deposit fee updated to: {:?}%.", self.deposit_fee_fund_manager);

        }

        //This method lets the fund manager trade with all the funds assests on whitelisted pools.
        //token_address is the asset you want to trade from.
        pub fn trade_radiswap(&mut self, token_address: ResourceAddress, amount: Decimal, pool_address: ComponentAddress){

            //checks if the pool is whitelisted
            let mut whitelisted=false;
            let defifunds: DefifundsComponent= self.defifunds.into();
            for (&address, &epoch) in defifunds.get_whitelisted_pool_addresses().iter(){
                if address == pool_address && epoch <= Runtime::current_epoch(){
                    whitelisted=true;
                }
            }
            assert!(whitelisted, "Trading pool is not yet whitelisted.");

            //do a trade using radiswap.
            let radiswap: RadiswapComponent = pool_address.into();
            let bucket_before_swap=self.vaults.get_mut(&token_address).unwrap().take(amount);
            let bucket_after_swap=radiswap.swap(bucket_before_swap);
            info!("You traded {:?} {:?} for {:?} {:?}.", amount, token_address, bucket_after_swap.amount(), bucket_after_swap.resource_address());

            self.add_token_to_fund(bucket_after_swap);

        }


    }
}
```

A.2 Defifunds - Blueprint

```
use scrypto::prelude::*;
use crate::fund::*;

blueprint! {


    struct Defifunds {
        funds: Vec<ComponentAddress>, //all funds in the Dapp
        defifunds_admin_badge: ResourceAddress,
        whitelisted_pool_addresses: HashMap<ComponentAddress, u64>, //whitelist valid from epoch <u64>
        defifunds_deposit_fee: Decimal,
        fee_vaults: HashMap<ResourceAddress, Vault>
    }

    impl Defifunds {

        pub fn instantiate_defifunds() -> (ComponentAddress, Bucket) {

            let defifunds_admin_badge: Bucket = ResourceBuilder::new_fungible()
                .divisibility(DIVISIBILITY_NONE)
                .metadata("name", "defifunds admin badge")
                .metadata("desciption", "Badge used for the admin stuff")
                .initial_supply(1);

            let access_rules = AccessRules::new()
                .method("new_pool_to_whitelist", rule!(require(defifunds_admin_badge.resource_address())))
                .method("remove_pool_from_whitelist", rule!(require(defifunds_admin_badge.resource_address())))
                .method("change_deposit_fee_defifunds", rule!(require(defifunds_admin_badge.resource_address())))
                .method("withdraw_collected_fee_defifunds", rule!(require(defifunds_admin_badge.resource_address())))
                .method("withdraw_collected_fee_defifunds_all", rule!(require(defifunds_admin_badge.resource_address())))
                .default(rule!(allow_all));

            let mut component = Self {
                funds: Vec::new(),
                defifunds_admin_badge: defifunds_admin_badge.resource_address(),
                whitelisted_pool_addresses: HashMap::new(),
                defifunds_deposit_fee: dec!(1),
                fee_vaults: HashMap::new()
            }
            .instantiate();
            component.add_access_check(access_rules);

            (component.globalize(), defifunds_admin_badge)

        }

        //fund make use of this method to deposit the fee to the correct vault
        //if other people decide to use this method it is just free money to the defifunds admin :D
        pub fn add_token_to_fee_vaults(&mut self, token: Bucket){
            let resource_address=token.resource_address();

            if !self.fee_vaults.contains_key(&resource_address){
                let key=resource_address;
                let value=Vault::new(resource_address);
                self.fee_vaults.insert(key,value);
            }

            self.fee_vaults.get_mut(&resource_address).unwrap().put(token);
        }

        ////////////////////////
        //methods for everyone//
        ////////////////////////

        pub fn new_fund(&mut self, fund_name: String, token: Bucket, initial_supply_share_tokens: Decimal) -> (Bucket, Bucket){
            let (fund, fund_manager_badge, share_tokens)=FundComponent::instantiate_fund(
                fund_name,
                token,
                initial_supply_share_tokens,
                Runtime::actor().as_component().0 //component address of Defifunds
            )
            .into();
            self.funds.push(fund.into());

            (fund_manager_badge, share_tokens)
        }

        pub fn get_fund_addresses(&mut self) -> Vec<ComponentAddress>{
            self.funds.clone()
        }

        pub fn get_defifunds_deposit_fee(&mut self) -> Decimal{
```

```
            self.defifunds_deposit_fee
        }

        pub fn get_whitelisted_pool_addresses(&mut self) -> HashMap<ComponentAddress, u64>{
            self.whitelisted_pool_addresses.clone()
        }



        ////////////////////////////
        //methods for defifund admin//
        ////////////////////////////

        pub fn new_pool_to_whitelist(&mut self, pool_address: ComponentAddress){
            self.whitelisted_pool_addresses.insert(pool_address, Runtime::current_epoch()+300); //valid after 300 epochs.
        }

        pub fn remove_pool_from_whitelist(&mut self, pool_address: ComponentAddress){
            self.whitelisted_pool_addresses.remove(&pool_address);
        }

        pub fn change_deposit_fee_defifunds(&mut self, new_fee: Decimal){
            self.defifunds_deposit_fee=new_fee;
        }

        pub fn withdraw_collected_fee_defifunds(&mut self, address: ResourceAddress) -> Bucket{
            self.fee_vaults.get_mut(&address).unwrap().take_all()
        }
        pub fn withdraw_collected_fee_defifunds_all(&mut self) -> Vec<Bucket>{
            let mut tokens = Vec::new();
            for vault in self.fee_vaults.values_mut(){
                tokens.push(vault.take_all());
            }
            tokens
        }



    }
}
```

A.3 Radiswap - Blueprint

This blueprint is made by members of the Radix team [40].

```
use scrypto::prelude::*;

blueprint! {
    struct Radiswap {
        /// The resource address of LP token.
        lp_resource_address: ResourceAddress,
        /// LP tokens mint badge.
        lp_mint_badge: Vault,
        /// The reserve for token A.
        a_pool: Vault,
        /// The reserve for token B.
        b_pool: Vault,
        /// The fee to apply for every swap
        fee: Decimal,
        /// The standard (Uniswap-like) DEX follows the X*Y=K rule. Since we enable a user defined
↪   'lp_initial_supply', we need to store this value to recover incase all liquidity is removed from the system.
        /// Adding and removing liquidity does not change this ratio, this ratio is only changed upon swaps.
        lp_per_asset_ratio: Decimal,
    }

    impl Radiswap {
        /// Creates a Radiswap component for token pair A/B and returns the component address
        /// along with the initial LP tokens.
        pub fn instantiate_pool(
            a_tokens: Bucket,
            b_tokens: Bucket,
            lp_initial_supply: Decimal,
            lp_symbol: String,
            lp_name: String,
            lp_url: String,
            fee: Decimal,
        ) -> (ComponentAddress, Bucket) {
            // Check arguments
            assert!(
```

```
            !a_tokens.is_empty() && !b_tokens.is_empty(),
            "You must pass in an initial supply of each token"
        );
        assert!(
            fee >= dec!("0") && fee <= dec!("1"),
            "Invalid fee in thousandths"
        );

        // Instantiate our LP token and mint an initial supply of them
        let lp_mint_badge = ResourceBuilder::new_fungible()
            .divisibility(DIVISIBILITY_NONE)
            .metadata("name", "LP Token Mint Auth")
            .initial_supply(1);
        let lp_resource_address = ResourceBuilder::new_fungible()
            .divisibility(DIVISIBILITY_MAXIMUM)
            .metadata("symbol", lp_symbol)
            .metadata("name", lp_name)
            .metadata("url", lp_url)
            .mintable(rule!(require(lp_mint_badge.resource_address())), LOCKED)
            .burnable(rule!(require(lp_mint_badge.resource_address())), LOCKED)
            .no_initial_supply();

        let lp_tokens = lp_mint_badge.authorize(|| {
            borrow_resource_manager!(lp_resource_address).mint(lp_initial_supply)
        });

        // ratio = initial supply / (x * y) = initial supply / k
        let lp_per_asset_ratio = lp_initial_supply / (a_tokens.amount() * b_tokens.amount());

        // Instantiate our Radiswap component
        let radiswap = Self {
            lp_resource_address,
            lp_mint_badge: Vault::with_bucket(lp_mint_badge),
            a_pool: Vault::with_bucket(a_tokens),
            b_pool: Vault::with_bucket(b_tokens),
            fee,
            lp_per_asset_ratio,
        }
        .instantiate()
        .globalize();

        // Return the new Radiswap component, as well as the initial supply of LP tokens
        (radiswap, lp_tokens)
    }

    /// Adds liquidity to this pool and return the LP tokens representing pool shares
    /// along with any remainder.
    pub fn add_liquidity(
        &mut self,
        mut a_tokens: Bucket,
        mut b_tokens: Bucket,
    ) -> (Bucket, Bucket) {
        // Get the resource manager of the lp tokens
        let lp_resource_manager = borrow_resource_manager!(self.lp_resource_address);

        // Differentiate LP calculation based on whether pool is empty or not.
        let (supply_to_mint, remainder) = if lp_resource_manager.total_supply() == 0.into() {
            // Set initial LP tokens based on previous LP per K ratio.
            let supply_to_mint =
                self.lp_per_asset_ratio * a_tokens.amount() * b_tokens.amount();
            self.a_pool.put(a_tokens.take(a_tokens.amount()));
            self.b_pool.put(b_tokens);
            (supply_to_mint, a_tokens)
        } else {
            // The ratio of added liquidity in existing liquidty.
            let a_ratio = a_tokens.amount() / self.a_pool.amount();
            let b_ratio = b_tokens.amount() / self.b_pool.amount();

            let (actual_ratio, remainder) = if a_ratio <= b_ratio {
                // We will claim all input token A's, and only the correct amount of token B
                self.a_pool.put(a_tokens);
                self.b_pool
                    .put(b_tokens.take(self.b_pool.amount() * a_ratio));
                (a_ratio, b_tokens)
            } else {
                // We will claim all input token B's, and only the correct amount of token A
                self.b_pool.put(b_tokens);
                self.a_pool
                    .put(a_tokens.take(self.a_pool.amount() * b_ratio));
                (b_ratio, a_tokens)
            };
            (lp_resource_manager.total_supply() * actual_ratio, remainder)
        };

        // Mint LP tokens according to the share the provider is contributing
        let lp_tokens = self
```

```
                .lp_mint_badge
                .authorize(|| lp_resource_manager.mint(supply_to_mint));

            // Return the LP tokens along with any remainder
            (lp_tokens, remainder)
        }

        /// Removes liquidity from this pool.
        pub fn remove_liquidity(&mut self, lp_tokens: Bucket) -> (Bucket, Bucket) {
            assert!(
                self.lp_resource_address == lp_tokens.resource_address(),
                "Wrong token type passed in"
            );

            // Get the resource manager of the lp tokens
            let lp_resource_manager = borrow_resource_manager!(self.lp_resource_address);

            // Calculate the share based on the input LP tokens.
            let share = lp_tokens.amount() / lp_resource_manager.total_supply();

            // Withdraw the correct amounts of tokens A and B from reserves
            let a_withdrawn = self.a_pool.take(self.a_pool.amount() * share);
            let b_withdrawn = self.b_pool.take(self.b_pool.amount() * share);

            // Burn the LP tokens received
            self.lp_mint_badge.authorize(|| {
                lp_tokens.burn();
            });

            // Return the withdrawn tokens
            (a_withdrawn, b_withdrawn)
        }

        /// Swaps token A for B, or vice versa.
        pub fn swap(&mut self, input_tokens: Bucket) -> Bucket {
            // Get the resource manager of the lp tokens
            let lp_resource_manager = borrow_resource_manager!(self.lp_resource_address);

            // Calculate the swap fee
            let fee_amount = input_tokens.amount() * self.fee;

            let output_tokens = if input_tokens.resource_address() == self.a_pool.resource_address()
            {
                // Calculate how much of token B we will return
                let b_amount = self.b_pool.amount()
                    - self.a_pool.amount() * self.b_pool.amount()
                        / (input_tokens.amount() - fee_amount + self.a_pool.amount());

                // Put the input tokens into our pool
                self.a_pool.put(input_tokens);

                // Return the tokens owed
                self.b_pool.take(b_amount)
            } else {
                // Calculate how much of token A we will return
                let a_amount = self.a_pool.amount()
                    - self.a_pool.amount() * self.b_pool.amount()
                        / (input_tokens.amount() - fee_amount + self.b_pool.amount());

                // Put the input tokens into our pool
                self.b_pool.put(input_tokens);

                // Return the tokens owed
                self.a_pool.take(a_amount)
            };

            // Accrued fees change the raio
            self.lp_per_asset_ratio =
                lp_resource_manager.total_supply() / (self.a_pool.amount() * self.b_pool.amount());

            output_tokens
        }

        /// Returns the resource addresses of the pair.
        pub fn get_pair(&self) -> (ResourceAddress, ResourceAddress) {
            (
                self.a_pool.resource_address(),
                self.b_pool.resource_address(),
            )
        }
    }
}
```

A.4 Transaction Manifest

If you want to learn how the transaction manifest work you can read the radix docs [43].

Instantiate DefiFunds

```
#set fee
CALL_METHOD ComponentAddress("${acc1}") "lock_fee" Decimal("100");

#call instatiate fund function
CALL_FUNCTION PackageAddress("${pkg}") "Defifunds" "instantiate_defifunds";

#take resources from worktop back to acc
CALL_METHOD ComponentAddress("${acc1}") "deposit_batch" Expression("ENTIRE_WORKTOP");

```

Instantiate Radiswap pools

```
#set fee
CALL_METHOD ComponentAddress("${acc1}") "lock_fee" Decimal("100");

#create btc_usdt_pool
CALL_METHOD ComponentAddress("${acc1}") "withdraw_by_amount" Decimal("10") ResourceAddress("${btc}");
TAKE_FROM_WORKTOP_BY_AMOUNT Decimal("10") ResourceAddress("${btc}") Bucket("btc");
CALL_METHOD ComponentAddress("${acc1}") "withdraw_by_amount" Decimal("200000") ResourceAddress("${usdt}");
TAKE_FROM_WORKTOP_BY_AMOUNT Decimal("200000") ResourceAddress("${usdt}") Bucket("usdt1");
CALL_FUNCTION PackageAddress("${pkg}") "Radiswap" "instantiate_pool" Bucket("btc") Bucket("usdt1") Decimal("1000")
↪  "lp_btc_usdt" "lp_btc_usdt" "url" Decimal("0.0001");

#create eth_usdt_pool
CALL_METHOD ComponentAddress("${acc1}") "withdraw_by_amount" Decimal("100") ResourceAddress("${eth}");
TAKE_FROM_WORKTOP_BY_AMOUNT Decimal("100") ResourceAddress("${eth}") Bucket("eth");
CALL_METHOD ComponentAddress("${acc1}") "withdraw_by_amount" Decimal("100000") ResourceAddress("${usdt}");
TAKE_FROM_WORKTOP_BY_AMOUNT Decimal("100000") ResourceAddress("${usdt}") Bucket("usdt3");
CALL_FUNCTION PackageAddress("${pkg}") "Radiswap" "instantiate_pool" Bucket("eth") Bucket("usdt3") Decimal("1000")
↪  "lp_eth_usdt" "lp_eth_usdt" "url" Decimal("0.0001");

#create doge_usdt_pool
CALL_METHOD ComponentAddress("${acc1}") "withdraw_by_amount" Decimal("500000") ResourceAddress("${doge}");
TAKE_FROM_WORKTOP_BY_AMOUNT Decimal("500000") ResourceAddress("${doge}") Bucket("doge");
CALL_METHOD ComponentAddress("${acc1}") "withdraw_by_amount" Decimal("50000") ResourceAddress("${usdt}");
TAKE_FROM_WORKTOP_BY_AMOUNT Decimal("50000") ResourceAddress("${usdt}") Bucket("usdt2");
CALL_FUNCTION PackageAddress("${pkg}") "Radiswap" "instantiate_pool" Bucket("doge") Bucket("usdt2")
↪  Decimal("1000") "lp_doge_usdt" "lp_doge_usdt" "url" Decimal("0.0001");

#take resources from worktop back to acc
CALL_METHOD ComponentAddress("${acc1}") "deposit_batch" Expression("ENTIRE_WORKTOP");

```

Instantiate malicious pool

```
#set fee
CALL_METHOD ComponentAddress("${acc1}") "lock_fee" Decimal("100");

#create btc_usdt_pool
CALL_METHOD ComponentAddress("${acc1}") "withdraw_by_amount" Decimal("10") ResourceAddress("${btc}");
TAKE_FROM_WORKTOP_BY_AMOUNT Decimal("10") ResourceAddress("${btc}") Bucket("btc");
CALL_METHOD ComponentAddress("${acc1}") "withdraw_by_amount" Decimal("10") ResourceAddress("${usdt}");
TAKE_FROM_WORKTOP_BY_AMOUNT Decimal("10") ResourceAddress("${usdt}") Bucket("usdt1");
CALL_FUNCTION PackageAddress("${pkg}") "Radiswap" "instantiate_pool" Bucket("btc") Bucket("usdt1") Decimal("1000")
↪  "lp_btc_usdt" "lp_btc_usdt" "url" Decimal("0.0001");


#take resources from worktop back to acc
CALL_METHOD ComponentAddress("${acc1}") "deposit_batch" Expression("ENTIRE_WORKTOP");

```

Deposit USDT and DOGE

```
#set fee
CALL_METHOD ComponentAddress("${acc3}") "lock_fee" Decimal("100");

#take 100doge and 40usdt from acc3 to worktop
CALL_METHOD ComponentAddress("${acc3}") "withdraw_by_amount" Decimal("100") ResourceAddress("${doge}");
TAKE_FROM_WORKTOP_BY_AMOUNT Decimal("100") ResourceAddress("${doge}") Bucket("doge");
CALL_METHOD ComponentAddress("${acc3}") "withdraw_by_amount" Decimal("40") ResourceAddress("${usdt}");
TAKE_FROM_WORKTOP_BY_AMOUNT Decimal("40") ResourceAddress("${usdt}") Bucket("usdt");

#put tokens into fund
CALL_METHOD ComponentAddress("${fund}") "deposit_tokens_to_fund" List<Bucket>(Bucket("doge"), Bucket("usdt"));

#take resources from worktop back to acc
CALL_METHOD ComponentAddress("${acc3}") "deposit_batch" Expression("ENTIRE_WORKTOP");


```