A Source-Code-Based Taxonomy for Ethereum Smart Contracts

Completed Research Paper

Adrian Hofmann
University of Würzburg
Sanderring 2, 97076 Würzburg, DE
adrian.hofmann@uni-wuerzburg.de

Julian Kolb
University of Würzburg
Sanderring 2, 97076 Würzburg, DE
julian.kolb@uni-wuerzburg.de

Luc Becker
University of Würzburg
Sanderring 2, 97076 Würzburg, DE
luc.becker@stud-mail.uni-wuerzburg.de

Axel Winkelmann
University of Würzburg
Sanderring 2, 97076 Würzburg, DE
axel.winkelmann@uni-wuerzburg.de

Abstract

As blockchain gained a lot of attention in IS research since its emergence, development into networks and applications have made it extremely relevant for multiple industry branches. Yet observations show, that there remains a lack of in-depth knowledge and standardization, particularly in the field of blockchain applications, DApps. These DApps often consist of multiple smart contracts, used to automate different processes and the technical elements have so far remained unexplored in depth. In this paper we address this problem by creating a data-driven taxonomy of the technical elements of 150 smart contracts within 101 DApps following the approach of Nickerson et al. (2013). We identified 28 dimension and 64 characteristics in our technical and code-based taxonomy.

Keywords: Smart Contract, DApp, Taxonomy, Source Code Analysis, Blockchain

Introduction

Blockchain technology has emerged in multiple applications and thus became a disrupting technology in both information systems research and industry since its conceptualization and realization through Nakamoto (2008) at the end of the 2000s. Starting as a simple state-machine application (Bitcoin), blockchain quickly extended into an environment that allowed decentralized Turing-complete computations, today known as the Ethereum network. The already existing idea of so-called smart contracts published by Szabo (1997) was enhanced into decentralized smart contracts to become a substantial driver for automation. Despite repeatedly being confused with contracts in the legal sense, a smart contract may also execute arbitrary program code to carry agreements and their implications between contractors. Based on this, a smart contract built like an application and fitted with a user interface (front-end) is called a Decentralized Application (DApp) (Antonopoulos and Wood 2018).

In contrast to a regular app, a DApp runs its back-end code in a decentralized peer-to-peer (P2P) network instead of central servers. The underlying program is published on the blockchain as byte-code to ensure compatibility and acceptable performance. Once published, the smart contract code can not be changed, and errors can only be fixed by deploying a new code version. However, the old version will always be accessible. Therefore, programmers have to ensure contract correctness and avoid unforeseen functional issues (Zheng et al. 2020).

From a conceptual and practical viewpoint, creating and evaluating a smart contract is a complex task. To counteract these issues, developers rely on standardized functionalities that have proven to be efficient and secure. However, still among the most desired improvements for the Ethereum ecosystem, which is to date the most-used smart contract platform, are "more general-purpose libraries" and "more standard interfaces" (Zou et al. 2019). Therefore, the main concerns for developers are that it is hard to guarantee the security of smart contracts and the lack of powerful tools that support the development and testing of the smart contracts (Zou et al. 2019).

This paper aims to guide and standardize the development by identifying common patterns, functionalities, and their relations in state-of-the-art smart contracts, which can serve as a basis for discussion for the development of standards and libraries. Therefore, we analyze Ethereum smart contract development and categorize smart contracts based on mutual code patterns. We summarize the resulting constellations in the form of taxonomy to "provide a structure and an organization to the knowledge of a field, thus enabling researchers to study the relationships among concepts" (Nickerson et al. 2013). Understanding the code patterns and their relationship can help researchers and developers focus on specific areas, especially where standardization is lacking or external libraries are being widely used but not yet standardized.

Therefore, we focus on the following research question:

RQ: Which common code patterns are used to develop smart contracts, and which archetypes of smart contracts can be distinguished based on these patterns?

Several blockchains are using the Solidity programming language. However, we focus our research only on smart contracts deployed to the Ethereum network since the source code of individual smart contracts is mostly publicly available. While this poses a limitation on the generalizability of the results, we argue that for a first, focused discussion, the Ethereum blockchain is an ideal candidate, as it is the most popular blockchain for practitioners and researchers alike. Additionally, the Ethereum blockchain hosts smart contracts for a wide range of applications from the areas of decentralized finance (DeFi), games, collectible assets, and social networking. We contribute to smart contract research and development by answering our research question while creating a taxonomy of smart contracts to support structuring the scientific discussion.

This paper organizes as follows to answer the research question: In the next section, we present the previous work related to our research, followed by our research approach for the taxonomy derivation and evaluation in our future research. Subsequently, we introduce the processes of defining meta-characteristics and ending conditions, data collection, taxonomy building, and the final taxonomy. We then proceed with a cluster analysis to identify relations of common patterns and summarize them into archetypes of smart contracts. Lastly, we examine the primary findings, their implications for research and practice, limitations, and future research building on this study.

Foundations and Related Work

Due to the initial use case of Bitcoin as the first blockchain application, the blockchain landscape had mainly consisted of cryptocurrencies. This landscape diversified with the development of more complex blockchain use-cases and programmable smart contracts. The first attempt to structure the various potential use-cases emerged in March 2015, three months before the initial release of the Ethereum platform, ultimately leading to the first wave of DApps (Buterin et al. 2014; Glaser and Bezzenberger 2015). In the years to follow, more and more blockchain applications were published, which lead researchers to further structure those applications in some manner.

As of today, some taxonomies on blockchain platforms and applications already exist (Sarkintudu et al. 2018; Tasca and Tessone 2019; Wieninger et al. 2019). Sarkintudu et al. (2018) and Tasca and Tessone (2019) provided taxonomies on blockchain platforms. The identified characteristics ranged from fundamental features, such as the underlying consensus mechanism, to specific considerations, such as the programming language and feature-set for smart contracts running on the platform. The taxonomy of Tasca and Tessone (2019) is more detailed in these regards and can be adopted as a solid basis for designing novel blockchain platforms.

In more specialized insights, taxonomies are focusing on the consensus mechanisms and provide profound insights (Yeow et al. 2017). However, the researched consensus mechanisms are not limited to blockchain networks but also contain other forms of distributed ledger structures, such as directed acyclic graphs. To understand these structures and design options, we refer to the taxonomy of Ballandies et al. (2021). Here, the research focuses on the data structure, transaction structure, and consensus of distributed ledgers.

So far, the research on smart contracts taxonomies is scarce. Tönnissen and Teuteberg (2018) have built a smart contract taxonomy and identified nine dimensions closely related to the ones found in legal contracts based on literature. This taxonomy is geared towards conceptualizing a smart contract based on business requirements. In contrast to this approach, we analyze smart contracts on a source code level to support the construction of smart contracts on a technical level.

Smart Contracts on the Ethereum Blockchain are primarily programmed in Solidity, a high-level programming language similar to JavaScript, making the transition easy for web developers. While Solidity allows writing maintainable and understandable code, this code can not run directly on the Ethereum blockchain. To ensure compatibility and high performance, the source code is optimized and compiled into byte-code, that can be run on the Ethereum Virtual Machine (EVM). This compiled code gets submitted to the Ethereum network and published. After publishing, the byte-code can be viewed by anyone who holds a copy of the Ethereum ledger or uses a blockchain explorer. However, this compiled code is not human-readable, and users are reluctant to interact with smart contracts with obfuscated functionalities. Therefore, developers have the option to publish the human-readable smart contract so that users can transparently verify the claimed functionalities. In this paper, we aim to categorize smart contracts based on this published source code.

There have been other approaches to analyze smart contracts on a technical level. Though, their goal was not to classify and group code patterns. Wöhrer and Zdun (2018) for instance, utilized a similar source-code-based approach to analyze design and security patterns (Wohrer and Zdun 2018). However, the authors did not show how those are combined in real-life smart contracts. Bartoletti and Pompianu (2017) also propose a taxonomy focused on the contracts' usage area and application categories. The authors also suggest some high-level design patterns. However, from a technical perspective, the provided patterns and categories are too superficial to provide meaningful guidelines for developing secure smart contracts.

Methodology

Our research follows a two-step approach that combines the qualitative and quantitative research methods as illustrated in Fig. 1 (Bryman 2006). In the first two stages (A + B), we use an inductive taxonomy development approach according to Nickerson et al. (2013) to classify the properties and core elements of smart contracts. Especially technologies which are the main focus of research, such as blockchain technology, can be better explained and understood with the help of taxonomies (Oberländer et al. 2019). Taxonomies offer a set of dimensions with differentiated and unique characteristics, with each entity having exactly one suitable attribute for each dimension (Nickerson et al. 2013). To create a rigorous taxonomy, we followed the seven steps framework published by Nickerson et al. (2013), which is well established within this area of information systems research (Fellmann et al. 2018; Rizk et al. 2018; Tönnissen and Teuteberg 2018). However, even with this approach, there is no guarantee that this is the optimal taxonomy since the research process is characterized by qualitative influences and by subjective decisions of the researchers (Nickerson et al. 2013).

In the first stage (A1-A4), we already used a limited set of data to develop an initial version of our taxonomy for gambling smart contracts (Kolb et al. 2020) to outline the topic. Our previous paper analyzed the smart contracts of gambling DApps to identify 18 dimensions and 41 characteristics of your technical and code-based taxonomy of gambling smart contracts. We chose these types of contracts because they can utilize diverse functionalities and standards. The analysis provided an ideal starting point to validate our approach of analyzing source code to categorize smart contracts. In fact, in this first research, we could identify many fundamental concepts still present in our final taxonomy, such as token standards, the usage of helper functions, or ownership handling of contracts (Kolb et al. 2020).

The first step was a rigorous and reliable data collection process (A1). We then determined the meta-

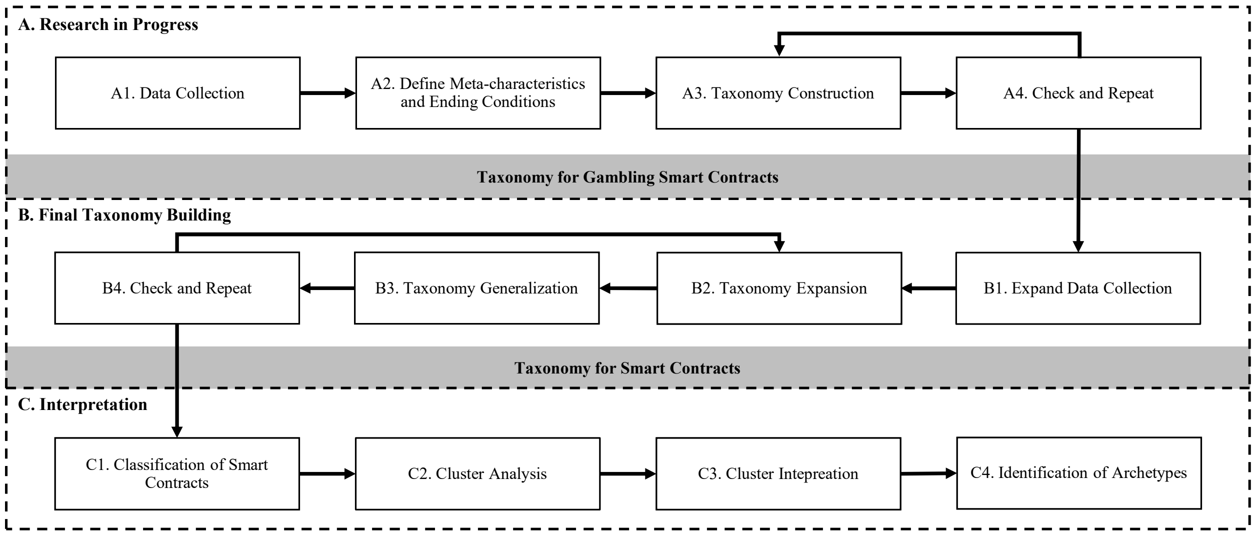
A. Research in Progress
A1. Data Collection
A2. Define Meta-characteristics and Ending Conditions
A3. Taxonomy Construction
A4. Check and Repeat
Taxonomy for Gambling Smart Contracts
B. Final Taxonomy Building
B4. Check and Repeat
B3. Taxonomy Generalization
B2. Taxonomy Expansion
B1. Expand Data Collection
Taxonomy for Smart Contracts
C. Interpretation
C1. Classification of Smart Contracts
C2. Cluster Analysis
C3. Cluster Intepreation
C4. Identification of Archetypes

Figure 1. Overview of research methodology

characteristics and the necessary ending conditions (A2). In the third step, the actual taxonomy was defined (A3). According to the taxonomy building guidelines of Nickerson et al. (2013), we used an iterative process to create, check, and modify dimensions and characteristics until the satisfaction of ending conditions (A4). We only outline these steps in this paper. For a detailed description, we refer to Kolb et al. (2020).

It is now necessary to provide general guidance to contribute to the development and design of smart contracts in business usage. In the paper at hand, we expand the data to a broad set of different smart contracts. To achieve this, we extend and finally generalize the taxonomy still following the same guidelines of Nickerson et al. (2013) as shown in Figure 1 (stage B). In stage C, we extend the purely descriptive analysis by clustering the smart contracts and identifying archetypes. To do so, we classified the smart contracts at hand with the final taxonomy (C1) and afterwards performed a cluster analysis to identify different archetypes to determine similarities and overarching patterns (C2). We then analyzed the clusters and interpreted the findings (C3) to ultimately identify archetypes of smart contracts (C4). These archetypes should help researchers and practitioners to understand the smart contract landscape better. For example, developers who want to build new Ethereum applications can use the archetypes to classify their applications and use the taxonomy to guide best practices for applications in these categories. Furthermore, it guides in identifying possibilities to improve the tools, libraries, standards, and techniques currently used to develop smart contracts on the Ethereum blockchain by highlighting the most used functionalities, which still lack standardization or toolsets. Ultimately, future developers can use our findings to streamline their results and efficiently learn from other projects.

A Taxonomy for Smart Contracts

The following section describes our process of taxonomy building. The final taxonomy will be presented in the section ahead.

Data collection

Our data collection process involves identifying appropriate resources and tools for gathering the data itself. We have conducted our inductive research approach by collecting data from the following databases: www.dapp.com and www.stateofthedapps.com. Both databases offer to filter by blockchain technology (Ethereum, EOS, Steem, ...) and categorization of the listed DApps. To limit our dataset, we filtered it for Ethereum based applications. In contrast to previous research, we did not limit our research to one category but analyzed all available categories (art, exchange, finance, gambling, game, high risk, tools, social, and others). We then added the to our previous gambling dataset. In the next step, we excluded all DApps that had no user activity in mid-2020 to ensure the current significance of the data. After this step, the dataset consisted of 183 DApps. Ultimately only DApps that allowed access to the underlying source code were taken into account. Although the other smart contracts are available as byte code on the Ethereum blockchain, this code is difficult to analyze as coherent wording, crucial to the researchers' understanding, is missing. While there are methods to reverse engineer and decompile these smart contracts, they have not proven reliable enough for rigor analyses (eevm 2020). This is a limitation of the research at hand, which can not be circumvented in the foreseeable future. The whole process results in a final dataset of 101 DApps, as depicted in Figure 2.

It should be noted that these DApps are often based on more than one smart contract. For the taxonomy, we treat these as one single contract. The reason for splitting the logic into multiple contracts that interact with each other can be easier readability and maintainability of the source code. The set of source code documents analyzed consisted of 150 source code files.

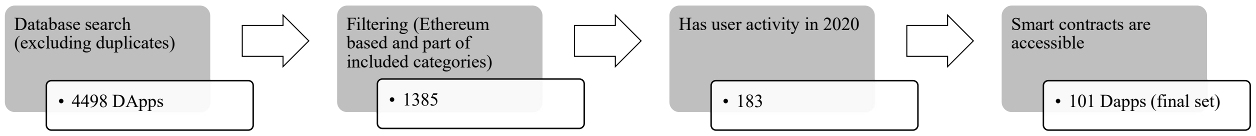
Database search (excluding duplicates)
• 4498 DApps
Filtering (Ethereum based and part of included categories)
• 1385
Has user activity in 2020
• 183
Smart contracts are accessible
• 101 Dapps (final set)

Figure 2. Data collection process

Meta-characteristics definition and ending conditions

The second step will define the scope, goals, meta-characteristic, ending conditions, and the taxonomy structure itself. The intention is to structure the utilized smart contracts to identify core patterns of the source code. The ultimate goal is to provide the necessary groundwork for further standardization and development of future smart contracts. The meta-characteristic of the taxonomy is *smart contract classes, functions (i.e. procedures) and code patterns*, and it can be directly derived from our research question.

The ending conditions for the taxonomy development process are divided into objective and subjective ending conditions (Nickerson et al. 2013). The objective conditions are:

OC1 - A representative sample of objects has been examined. The set of 101 smart contracts is small compared to the estimated one million smart contracts currently deployed on the Ethereum network. However, we argue that the sample is on the upper limit of what is reasonable to analyze. Additionally, to ensure that the sample is representative, we used a stricter objective condition 2.

OC2- In the last five iterations, no characteristics or dimensions are combined, divided, or added. While Nickerson et al. (2013) suggest that in the *last* iteration, no dimensions should be combined, divided, or added, we expand this to the last five iterations. We do so to counteract the possibility of having a too-small sample. We examined a set of 5 smart contracts in each iteration, so if the taxonomy changes in the last five iterations, we have to expand our dataset by including less popular smart contracts. If the taxonomy does not change for five iterations, we can be confident that the dataset is representative enough.

OC3 - At least one object is classified under every characteristics of every dimension. We ensured that this ending condition is met after each iteration by only adding characteristics, with at least one object classified.

OC4 - Every dimension is unique and not repeated, every characteristic is unique within its dimension. Again, we ensured that this ending condition was met after each iteration. While we do have some characteristics, that are not unique (e.g., *implemented* and *Non-Implemented*), they are unique within their respective dimension.

The subjective conditions were, that the taxonomy has to be *concise*, *robust*, *comprehensive* and *extendable*.

SC1 - Concise. This condition encourages the taxonomy to be meaningful without being overwhelming. Note that Nickerson et al. (2013) postulated 5-9 dimensions as an adequate range (Nickerson et al. 2013). While our taxonomy exceeds these recommendations, it was not justifiable to delete or combine dimensions in the last iterations. We argue that this extensive taxonomy is necessary to grasp the possible granularity of design decisions on a code-level basis.

SC2 - Robust. The characteristics have to provide a sufficient differentiation among the objects. Differentiation is an important condition since it is important for clustering. If the characteristics do not differentiate enough, the clustering will not be meaningful.

SC3 - Comprehensive. All objects within the domain of interest must be able to be classified, all dimensions of interest must be identified. This condition is linked to OC2. We ensured that each new smart contract could be classified completely with the current state of the taxonomy for each iteration.

SC4 - Extendable. New dimensions and characteristics can be added to the taxonomy. While taxonomy is already extensive, there is still room to extend it. There is a constant stream of new applications deployed to the Ethereum blockchain with new trends and standardization efforts. When new patterns in smart contracts gain popularity, they can be added easily to the current taxonomy.

Taxonomy Building

After having defined the meta-characteristics and the ending conditions, we started to extend our taxonomy. The source code of the smart contracts was queried from Etherscan[1] and stored as text documents. The analysis of the source code was twofold: two researchers started highlighting the functions in every contract. While doing so, they derived a coding scheme in an iterative procedure using MAXQDA as supporting software. The coding scheme was only a supporting tool to enable the two researchers to have a standardized way to label functions and classes according to their purpose in the source code. Therefore, the coding system was not checked for inter-coder reliability. We then compared the results of both researchers and discussed them after each iteration within a panel of experts.

Overall we analyzed over 120,000 lines of source code and coded over 1,100 passages in this source code with a final set of 70 different codes. After each iteration, similar codes were grouped if possible and large coding categories were split if possible. Additionally, each researcher created a category "unknown" in each iteration where he marked passages in the source code that were not similar to the other categories or where the functionalities were unclear. Both researchers analyzed these together to decide how they should be labeled. The codes from the labeling process were used to derive the dimensions and characteristics of the taxonomy. Characteristics were added, removed, or merged depending on the number of occurrences in the smart contracts and their importance in the source code. This process was guided by the subjective conditions SC1-SC4.

Smart Contract Taxonomy

Our analysis of 101 smart contracts yielded 64 characteristics from 28 dimensions, grouped into six categories. The complete taxonomy is presented in Figure 4 and is described in the following chapter.

DApp Design

The category DApp Design contains dimensions and characteristics related to the basic architecture of the examined application. Our research has shown that DApps differ in their quantity of employed smart contracts, which we describe as *smart contract quantity*. Our dataset identified two distinct types of applications: DApps that combine all functionalities in one single smart contract and DApps that split functionality among multiple smart contracts.

Dimensions	**Characteristics**	
Smart Contract Quantity	Single	Multiple

Table 1. Dimensions and characteristics of *DApp Design*

[1]https://etherscan.io/ - Ethereum (ETH) Blockchain Explorer

Core Functionality

In the second category, we examined how smart contracts process their actual core functionality in the smart contracts and defined it as Core Functionality.

Most smart contracts within our research are including, what we call *Core Logic* within their code. These functions comprise game rules in gambling smart contracts or auction mechanisms in exchange or finance DApps. Other applications exclude these functionalities and access external software via various interfaces, returning results to the smart contract.

Smart contracts also deal differently with their *Usage Fee*. Some smart contracts provide a changeable fee, which the owner or administrator of the contract may modify. Other applications use an initially fixed usage fee or do not use such a fee at all.

When we first looked at gambling smart contracts, *Asset Handling* was a big issue there. However, this functionality is also widely used in other application areas and is part of the core functionality. Asset Handling mechanisms help to manage, transferring or proving ownership of various tangible or intangible objects. Those can be digital objects in games, real estate, or share certificates, among others. We could differentiate between smart contracts, which use transferable assets, or which only provide non-transferable assets.

Dimensions	**Characteristics**		
Usage Fee	Changeable	Fix	None
Core Logic	Included	Excluded	
Asset Handling	Transferable	Non-transferable	

Table 2. Dimensions and characteristics of *Core Functionality*

Helpers

Many smart contracts require augmented features on top of the implemented core functionality. These features can be hard to implement within the contract, or sometimes data outside the smart contract needs to be accessed. For this purpose, many smart contracts include libraries, which provide various functions. In total, we have identified five different helper libraries frequently used in smart contracts: *Byte Helpers*, *String Helpers*, *Math Helpers*, *Oracles*, and *Interfaces*. While the helper functions provide functionalities that programmers are familiar with from other programming languages, Oracles and Interfaces are different. The former are used to retrieve data from non-blockchain sources. The ladder are used to interact with the smart contract from outside the blockchain.

Math Helper libraries provide different functionality regarding mathematical operations. From simple functions that determine the minimum or maximum of two (or more) numbers, those libraries may extend to functions that implement square root, logarithmic, or exponential functions. Since these are based solely on the limited mathematical capabilities of the Ethereum byte-code, they often require an iterative approach to calculate the desired value based on the basic mathematical operations (division, multiplication, addition, and subtraction). This approach makes some of the functions costly to execute. Correspondingly, some math libraries carry as annotation the following warning: “This is where your gas goes.”.

Similarly, *String Helpers* provide functionalities used to manipulate strings. Among the most common functions implemented are checking the lengths of a string, slicing it into sub-strings, and concatenating or comparing two strings. Since Ethereum does not provide a string datatype, the bytes datatype is used for this purpose. The bytes are always interpreted as UTF-8 encoded strings in the contracts we examined.

Byte sequences that do not represent strings but store arbitrary data are also common to the contract code. *Byte Helper* libraries are used to manage this very flexible data type. Like the string functions, they often allow slicing and concatenating byte sequences, but they are also used to transform bytes into other data types like unsigned integers or Ethereum addresses. Unlike the previous libraries, the Byte functions often use inline assembly code to manipulate storage efficiently.

An *Oracle* is a service that allows importing data into a DApp or smart contract from an external source like the Internet (Xu et al. 2016). These Oracles are mainly used to query the results of external events (like sports matches, real estate data, or stock exchanges) or to include an external source of randomness (especially in gambling and high-risk contracts). Querying Oracles is possible through services like Provable™, that supply their libraries to interact with the Oracles (Provable 2020). Oracles have some criticism since they rely on a centralized source of truth, which can be manipulated on an otherwise very secure network.

Interfaces are needed to interact with smart contracts from outside the blockchain. While every smart contract provides callable public functions, certain conventions allow interaction with a contract in a standardized way. While many token standards provide their standardized interface, querying whether a contract supports or not is difficult. Therefore, the ERC165 standard can be used to query a specific contract for its available standard interfaces.

Dimensions	**Characteristics**	
Math Helpers	Implemented	Non-implemented
String Helpers	Implemented	Non-implemented
Byte Helpers	Implemented	Non-implemented
Interfaces	ERC16	Others
Oracles	Implemented	Non-implemented

Table 3. Dimensions and characteristics of *Helpers*

Contract Management

We noticed that smart contracts differ very clearly in their ability to be controlled. Among others, this includes roles, ownership handling, rebranding, updating, and killing smart contracts. We have summarized dimensions in this area in the Contract Management category.

First, we propose a differentiation between different *Roles*. The roles *Token Owner* and *Admin* are usually available. Both can occur alone or in combination. In addition, some smart contracts define individual roles, which we have not included as a further characteristic because they differ from application to application. Using roles permits the smart contracts to control various user functionalities: the right to execute transactions, change the contract, or generally access certain functions. Some smart contracts may also not use roles at all.

In addition to the Admin and Token Owner roles already introduced, there is the contract owner. The owner of a smart contract has significant rights, like killing and pausing a contract. We identified various ways of how smart contracts deal with *Ownership Handling*. First, some smart contracts do not allow ownership and are therefore non-ownable. Opinions differ here as to whether an owner has a positive or negative effect on a smart contract. However, we observed that about one-third of the smart contracts do not utilize the implementation of an owner and thus fully support the principle of decentralization in a blockchain. We also found out that not all smart contracts provide a function to transfer ownership. This can have multiple reasons, yet the predominant is that a smart contract should not be transferred at all. We also investigated that in some contracts, a transfer can be renounced or must be actively accepted. We did not expect this functionality at the beginning of our investigation and were surprised about this feature. We assume that it is used as a security feature preventing an accidental transfer of ownership in some cases.

Another dimension in the Contract Management category is the ability to *Rebrand* a smart contract. Only 5% of the examined smart contracts implemented this function, but we are very critical about it. If such a possibility is implemented, the admin or owner can rename the smart contract and present it differently to the outside world. We assume that this function is used mainly by dubious applications, allowing scamming more users under different names.

While the contract code is unchangeable once a smart contract has been deployed to the blockchain, there still exist ways to make the smart contract *Upgradable*: Via proxy contracts. They delegate the contract call to the current version of the contract. If a new version is deployed, a variable in the proxy contract is changed to the address of the new version.

Finally, we distinct between smart contracts that are *Killable* and those that are not. About 15% of the examined smart contracts provide a function, which specifies the end of life of a smart contract and ends all pending transactions.

Dimensions	**Characteristics**			
Roles	Admin and Token owner	Token owner	Admin	None
Ownership Handling	Non-ownable	Ownable, Transferable and Non-renouncable	Ownable, Transferable and Renouncable	Ownable and Non-transferable
Rebrandable	Rebrandable		Non-rebrandable	
Upgradable	Upgradable		Non-upgradable	
Killable	Killable		Non-killable	

Table 4. Dimensions and characteristics of *Contract Management*

Safety Functions

Since smart contracts are characterized by a decentralized organization and usually do not have a central controlling authority, some Safety Functions are necessary to ensure proper operation. Their use can be to check transactions and validate them first to prevent false and inadvertent transactions. Some functions are employed to control and limit the influence of individuals within the system.

A *Check Address* function is used in smart contracts to check the validity of wallet addresses and other smart contracts before executing a function. This check can prevent permanent loss by transferring user tokens or other objects to an invalid address or smart contract. These functions can be either implemented or not.

The official Solidity guidelines recommend implementing a *Default Function* in a smart contract. This function gets executed when sending a transaction to the smart contract without input data and can be used to load funds into a contract. Additionally, it is often used as a safety measure to prevent users from accidentally sending Ether to the contract. In this case, the default function reverts the transaction. Often, however, the default function is implemented without any additional functionality.

By using the *Default Function*, non-specific requests to the smart contract are processed and, if necessary, rejected. Some smart contracts implement them without additional logic, while some throw error messages or execute supplementary code. Sometimes, a Default Function is not implemented at all.

Contrary to the *Math Helpers*, the *Safe Math* functionality is present in most contracts that require even the most simple calculations. The basic math functions in Ethereum do not check for overflow or underflows of the variables and can therefore yield wrong results. *Safe Math* functions monitor additions, subtractions, multiplications for overflows and underflows while additionally implementing integer division.

In 2016 over 3.6 million Ether were stolen when the contract of the popular DApp TheDAO was hacked. The attackers used a reentrancy attack to funnel the funds out of the contract. As a countermeasure, many smart contracts implement a *Reentrancy Guard*, that prevents this type of attack.

Especially token sales try to prevent single users from acquiring lots of tokens in an early sale stage. These measures are summarized as *Anti Early Whale* protocols.

If a contract shows unexpected behavior or is under attack by a malicious party, some contracts have a *Pause Contract* functionality. This function can pause and unpause the complete functionality of the contract until normality is restored.

Another method to deal with unexpected behavior is the option to *Refund Users*. Users can request a refund that is to be approved by contract owners or administrators if the request is justified.

In some cases, a transaction may get stuck in an automated contract (pending) and cannot be processed further. In this case, around 20% of the contracts in our dataset offer the possibility to *Withdraw Pending Transactions* to the user. In this case, the transaction is cancelled, and affected tokens are credited back to the user.

Dimensions	Characteristics		
Check Address	Implemented		Non-implemented
Default Function	Implemented	Additional Logic	Non-implemented
Refund User	Implemented		Non-implemented
Safe Math	Implemented		Non-implemented
Reentrancy Guard	Implemented		Non-implemented
Anti Early Whale	Implemented		Non-implemented
Pause Contract	Implemented		Non-implemented
Withdraw Pending Transaction	Implemented		Non-implemented

Table 5. Dimensions and characteristics of *Safety Functions*

Tokens

Tokens are the components that create an economic incentive in blockchain technology. They are units of local values and are primarily used to foster the operation of a blockchain (Shin et al. 2019). For example, nodes in a blockchain receive rewards for their work in the network using different protocols. In some cases, some marketplaces allow these tokens to be traded with each other and being exchanged for fiat money, thereby assigning them a monetary value (Hülsemann and Tumasjan 2019).

These tokens can be characterized by different criteria, as demonstrated in the following. For example, Euler (2021) and Hülsemann and Tumasjan (2019) divide the intention of token use into cryptocurrencies, network tokens, and investment tokens. In our dataset, we could not detect any significant differences. Therefore, we concentrate on the technical features (*Token Usage*). Hülsemann and Tumasjan (2019) classify them as blockchain-native tokens, non-native tokens, and DApp tokens. Within our analysis, however, we only recognized native tokens or DApp Tokens.

When implementing a token into a blockchain network, the developer can choose between different *Token Standards* or create a new one: The ERC20 standard served as the groundwork for more recent standards such as ERC223, ERC667, ERC721, and ERC777 developed within the Ethereum network (Victor and Lüders 2019). These differ according to various characteristics such as fungibility or other individual features. Among the examined DApps, the characteristics for the dimension token standard could be almost exclusively identified as ERC20 (40%) or ERC721 (10%) tokens. In our first study, which took place about six months earlier, the ERC721 standard was still very rare. In the meantime, however, it obviously established itself and is now used in about 10% of the contracts examined. Approximately 40% of the smart contracts studied do not use tokens at all. The remaining 10% either use Multiple standards or rely on a Modified token.

As a further dimension, we distinguished between Contracts where new tokens can be created or not. Mintable tokens allow the user to mint a token or stop the process, such as mint() and finishMinting(). Most mintable tokens are ERC20 based, which include an additional function that helps to increase the stock. This means that the supply is not fixed, although you can specify the initial stock level in the contract. According to our results, tokens can be mintable or are already created initially.

The opposite of *Mintable* tokens are tokens that are *Burnable*, which means that tokens can be destroyed and never used again. These two features do not exclude each other. While some contracts implement burnable tokens accidentally by allowing tokens to be sent to an invalid address, the *Burnable* feature is an intentional property of the token.

Finally, we can describe the characteristics of *Trading* and *Accounting* for the tokens in use. First, tokens can either be *sold* and bought, or the functionality is not implemented. Furthermore, either deposit and withdrawal are possible, or only withdrawal is permitted. As before, this functionality may even not be implemented at all.

Dimensions	Characteristics				
Token Usage	Native Token			DApp Token	
Token Standard	ERC721	ERC20	Multiple	Modified	None
Burnable	Burnable			Non-burnable	
Mintable	Mintable			Non-mintable	
Token Trading	Buy and Sell			None	
Token Accounting	Deposit and Withdrawal		Withdrawal only	None	

Table 6. Dimensions and characteristics of *Token*

Archetypes of Smart Contracts

Our descriptive analysis provides an overview of real-life examples of smart contract characteristics and code structures. However, it does not shed light on the combined characteristics of distinct types of smart contracts that act as boundary objects in the Ethereum smart contract ecosystem. Therefore, we performed a cluster analysis to determine these latent functional clusters of smart contracts that represent smart contracts archetypes. Additionally, a meaningful result of the clustering validates the meaningfulness of the taxonomy to some extent. We used agglomerative clustering hierarchical clustering (Wards Method) with the Jaccard distance because this method is well suited for clustering categorical data, as seen in the analysis of other taxonomies (Fischer et al. 2020; Gimpel et al. 2017). Choosing the appropriate number of clusters is always a challenging task. To support our decision, we analyzed three common metrics: the Elbow-Criterion (Madhulatha 2012), Silhouette coefficient (Devaraj et al. 2007) and Dunn index (Devaraj et al. 2007). Figure 3 displays the three scores concerning the number of clusters chosen. While according to the Elbow criterion, the optimal number of clusters should be five or seven, the other scores suggest a much higher number of clusters. As with previous research, there is a trade-off between interpretability and accuracy of the clustering. We, therefore, opted for seven clusters since it yields a suitable basis for interpretation as it shows a significant decrease in within-cluster variance. In our opinion, it represents the most comprehensive yet manageable solution to distinguish smart contracts.

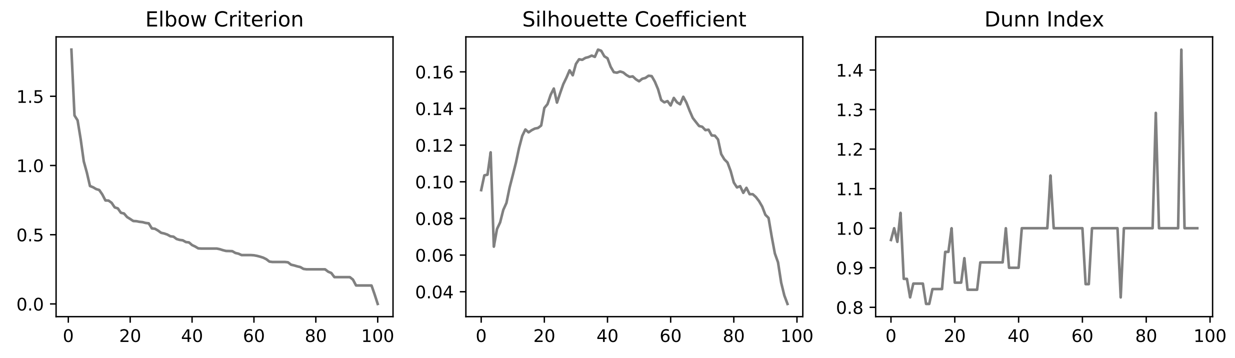

Figure 3. Clustering Scores for Optimal Choice of Clusters

Due to distinct functional differences, the number of observations comprised in each cluster varies significantly. The cluster sizes reach from 9 to 25 smart contracts per cluster. However, most hold between 10 to 15 contracts. In Figure 4 we summarize the clustering result and visualize the smart contract features in addition to the full taxonomy. In the rest of this section, we describe the identified clusters, draw conclusions on development structures and provide actionable insights for future smart contract development.

We describe the clusters not in the same order as they are shown in Figure 4 to highlight similarities and crucial differences between the clusters.

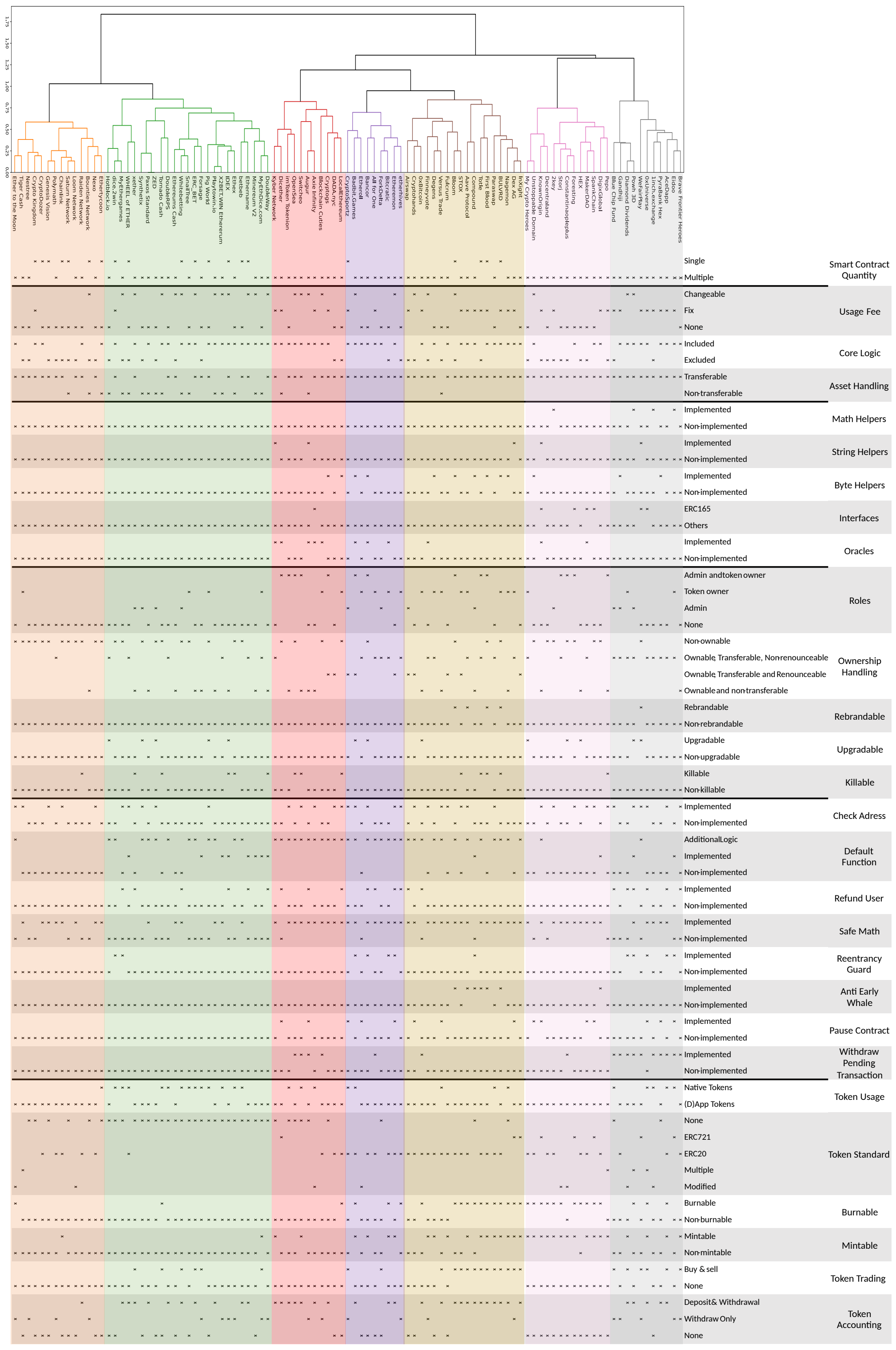

Figure 4. Clustering and Visualization of Smart Contract Characteristics

Archetype 4: "Bets" on Off-Chain Events

The contracts in this category have two notable similarities: using Oracles to interact with off-chain data and a fully implemented default function to receive Ether. The contracts are mostly classified into the *Gambling* and *Exchange* categories. The contracts allow bets on real-life events with Ether (may it be sports or financial bets).

While most contracts relied on a library for interaction with oracles, the specifics of the implementations often differed. As Zou et al. (2019) noted, developers wish for easier interaction with off-chain data. We agree that there is a lack of standardization. Notably, there exists an Ethereum Improvement Proposal (EIP) to tackle this issue and provide a standardized oracle interface (*Ethereum Improvement Proposals* 2021). It should be noted that two gambling contracts combined Oracles with string helper functions to process the retrieved data. This parallelism should be kept in mind when standardizing oracles so that developed string libraries or even native data types are compatible with the oracle standard.

It is noteworthy that the usage of oracles is somehow very limited to the use-case of betting. We are unsure about the reason behind this but hope that standardization can help adopt the concept to broader applications.

Archetype 2: Technically Secure Implementations of Financial Applications

This archetype includes mainly contracts that include functionalities of *initial coin offerings* (ICOs) and general token sales. There is a high usage of ERC20 tokens with the Burnable function (e.g., burning unsold tokens). Often they also include the functionality to mint additional tokens. However, the main functionality of these tokens is that they can be bought, sold, and transferred through the smart contract. While ICOs generally yield a financial risk for investors, the high usage of standardized functionalities, implementation of security features, and in some cases, usage of helper libraries indicates that developers and issuers of these ICOs are aware of risks. This use is undoubtedly motivated by the implications of losing the revenue of the ICO due to technical errors.

Here, we can show that there is enough knowledge in the developer community to build secure financial applications. However, the knowledge should be accumulated to implement standard libraries for this security functionality.

Archetype 1: Tokenized Asset Contracts without user management

The tokens in this cluster all used one or multiple token standards. However, the tokens can not be bought, sold, deposited, or withdrawn, which is the main differentiator between this Archetype and Archetype 2. A similarity, however, is that almost all of them have the functionality to mint or burn tokens. In this case, this is to create or destroy assets in an otherwise self-contained system. Half of them implement the functionality to check the recipient address to prevent accidentally losing assets by transferring them to an invalid address. The contracts track things like game assets or register (domain) names on the blockchain.

Archetype 3: Asset Centered Contracts with User Management

This cluster is very similar to Archetype 1. However, in this cluster, most contracts handle their ownership, and the roles of users. The question arises whether these functionalities were not included in contracts of Archetype 1 because they do not provide an additional benefit or are too complex to implement for added value. It can be hypothesized that if ownership and role management are standardized, Archetypes 1 and 3 merge into one since their functionalities get implemented in almost every contract. Here, further research on the standardization of governance in Ethereum blockchains is needed to provide a viable solution. We propose to conduct further research in this area along with the research framework for governance in blockchains developed by Beck et al. (2018).

Archetype 0: High Value Asset Management

The smart contracts in this cluster have much functionality regarding the ownership of the contract. Additionally, some of them provide role management. The rest of the functionality is quite mixed, and the cluster is quite unstructured. The contracts in this cluster use many tokens. However, they are not standardized. However, almost all of them offers users the functionality to withdraw pending transactions, a feature that is rarely seen in other archetypes. The contracts often handle collectables. Therefore, losing one because of a faulty transaction or insufficient gas can yield high losses. We argue that the functionality of withdrawing pending transactions should be implemented more often with non-fungible tokens such as ERC721, which can represent ownership of valuable assets.

Archetype 5: Simple Contracts and Miscellanous

This archetype does not offer much standard functionality with other contracts. Contracts of this archetype do not implement any helper functions, role management and rarely any standardized function. This can either mean that the contracts have a simple structure. However, many contracts are comprised of multiple contract files. Therefore, this cluster is a catchall category for otherwise uncategorized contracts.

Archetype 6: Simple Contracts and Miscellaneous with Asset-Based Governance

Like Archetype 5, this category is not as clearly defined as the other archetypes as very few patterns are present in this cluster. However, unlike Archetype 5, the contracts employ more asset handling functionalities and define roles for token owners. The core functionality of some contracts is far from standard functionalities. For example, solutions that increase the transaction rate on the Ethereum blockchain or make it interoperable with other blockchains are in this category. Standardization of these contracts is quite challenging. However, these are rare use-cases, and we, therefore, do not see a necessity for standardization.

Conclusion

Smart contracts and DApps are part of the disruptive blockchain technology, facilitating peer-to-peer transactions and decentralized applications. They have, therefore, become an increasingly important topic in information systems research. However, their technical and functional characteristics are not yet well understood and not yet standardized. To structure the smart-contract landscape on a technical level, we developed a source-code-based taxonomy for Ethereum smart contracts.

We used a data-driven method of taxonomy building to provide descriptive knowledge and a structure that had been missing in the previous discussion and development of smart contracts. We, therefore, collected empirical data on 101 DApps comprising 150 smart contracts and used an iterative research approach following Nickerson et al. (2013). This approach led us to define a final taxonomy consisting of 28 dimensions with 64 characteristics based on the six meta-categories. As defined by its nature, a taxonomy is never complete and should be expandable in its dimensions and characteristics as new objects emerge (Nickerson et al. 2013). However, it turned out that challenging our first taxonomy, which focused on applications in gambling, only resulted in minor changes and extensions. No new meta-categories were created or significantly changed. After the final taxonomy was created, we classified the examined DApps for evaluation. All 101 DApps could be fully mapped in the taxonomy. We then clustered the smart contracts and identified seven archetypes of smart contracts. We could identify some patterns that show room for improvement, especially regarding standardization and the development of additional libraries.

By providing a taxonomy for smart contracts, which can be used for future research and development projects, we contribute to current research and practice. Uncovering the technical characteristics within usage categories may help researchers better classify and analyze smart contracts in depth. New blockchain applications can now be conceptualized on a platform level using higher-level taxonomies such as the one from Tasca and Tessone (2019). Then the smart contracts can be conceptualized on a business level based on the work of Tönnissen and Teuteberg (2018). Finally, our taxonomy can then be used to specify technical details for the programmers.

By analyzing the clusters, we showed that applications have strong technical similarities while sometimes serving different purposes. The development of new contracts can use our results to consider architectural designs and standardize functionalities. For example, we identified similar functionalities used by secure financial applications that could be merged into a financial application framework. This would enable programmers to bundle their resources and reuse secure functionalities.

Qualitative research work such as source code analysis is usually subject to the subjectivity of the researchers involved. While source code does not leave much room for interpretation, we have considered this problem by integrating additional researchers into the process. The coding was carried out, verified, compared and discussed by a total of four people. Nevertheless, even after this generalization of the taxonomy, it is still possible that our coding is incomplete, incorrect or subjective.

Additionally, this research is focused on only one blockchain network. Some challenges go beyond the Ethereum ecosystem. Therefore, we suggest expanding the research to other platforms. It should be started with technologies that also use the Ethereum Virtual Machine as a technological basis, such as the Ethereum test networks, the fork Ethereum Classic or other unrelated networks like Avalanche Ecosystem (Rocket 2018). By doing so, many patterns and standards can be reused. In a second step, the research should be expanded to technologies that use different programming languages. Adapting the research methodology and concepts to these platforms is challenging but is needed to lay a basis for cross-chain standardization and tooling.

Acknowledgement

This work has been developed in the project PIMKoWe. PIMKoWe (reference number: 02P17D160) is partly funded by the German ministry of education and research (BMBF) within the research programme "Industrie 4.0 – Kollaborationen in dynamischen Wertschöpfungsnetzwerken (InKoWe)" and managed by the Project Management Agency Karlsruhe (PTKA). The author is responsible for the content of this publication.

References

Antonopoulos, A. M. and Wood, G. 2018. *Mastering ethereum: building smart contracts and dapps*. O'reilly Media.

Ballandies, M. C., Dapp, M. M., and Pournaras, E. 2021. "Decrypting distributed ledger design—taxonomy, classification and blockchain community evaluation". *Cluster Computing* (2021).

Bartoletti, M. and Pompianu, L. 2017. "An empirical analysis of smart contracts: platforms, applications, and design patterns". In: *International conference on financial cryptography and data security*. Springer, pp. 494–509.

Beck, R., Müller-Bloch, C., and King, J. L. 2018. "Governance in the blockchain economy: A framework and research agenda". *Journal of the Association for Information Systems* (19:10), p. 1.

Bryman, A. 2006. "Integrating quantitative and qualitative research: how is it done?" *Qualitative research* (6:1), pp. 97–113.

Buterin, V. et al. 2014. "A next-generation smart contract and decentralized application platform". *white paper* (3:37).

Devaraj, D., Valarmathi, K., Kanmani, J., and Radhakrishnan, T. 2007. "Hybrid GA Fuzzy Controller for pH Process". In: *Computational Intelligence and Multimedia Applications, International Conference on*. Vol. 2. Los Alamitos, CA, USA: IEEE Computer Society, pp. 13–18.

eevm 2020. *Panoramix*. https://github.com/eveem-org/panoramix.

Ethereum Improvement Proposals 2021. [Online; accessed 4. May 2021].

Euler, T. 2021. *The Token Classification Framework: A multi-dimensional tool for understanding and classifying crypto tokens. – Untitled INC.* [Online; accessed 4. May 2021].

Fellmann, M., Koschmider, A., Laue, R., Schoknecht, A., and Vetter, A. 2018. "Business process model patterns: state-of-the-art, research classification and taxonomy". *Business Process Management Journal* (25:5), pp. 972–994.

Fischer, M., Heim, D., Hofmann, A., Janiesch, C., Klima, C., and Winkelmann, A. 2020. "A taxonomy and archetypes of smart services for smart living". *Electronic Markets* (30:1), pp. 131–149.

Gimpel, H., Rau, D., and Röglinger, M. 2017. "Understanding FinTech start-ups – a taxonomy of consumer-oriented service offerings". *Electronic Markets* (28:3), pp. 245–264.

Glaser, F. and Bezzenberger, L. 2015. "Beyond cryptocurrencies-a taxonomy of decentralized consensus systems". In: *23rd European conference on information systems (ECIS), Münster*. Münster.

Hülsemann, P. and Tumasjan, A. 2019. "Walk this Way! Incentive Structures of Different Token Designs for Blockchain-Based Applications". In: *Fortieth International Conference on Information Systems*. Munich.

Kolb, J., Hofmann, A., and Becker, L. 2020. "Building a Taxonomy for Gambling Smart Contracts". In: *European Conference on Information Systems (ECIS)*. Marrakesh.

Madhulatha, T. S. 2012. "An overview on clustering methods". *arXiv preprint arXiv:1205.1117* ().

Nakamoto, S. 2008. *Bitcoin: A peer-to-peer electronic cash system*.

Nickerson, R. C., Varshney, U., and Muntermann, J. 2013. "A method for taxonomy development and its application in information systems". *European Journal of Information Systems* (22:3), pp. 336–359.

Oberländer, A. M., Lösser, B., and Rau, D. 2019. "Taxonomy research in information systems: A systematic assessment". In: *Proceedings of the 27th European Conference on Information Systems (ECIS)*. Uppsala.

Provable 2020. *Provable - Blockchain Oracle Service, Enabling Data-Rich Smart Contracts*.

Rizk, A., Bergvall-Kåreborn, B., and Elragal, A. 2018. "Towards a taxonomy for data-driven digital services". In: *Proceedings of the 51st Hawaii International Conference on System Sciences*.

Rocket, T. 2018. "Snowflake to avalanche: A novel metastable consensus protocol family for cryptocurrencies". *Available [online].[Accessed: 4-12-2018]* ().

Sarkintudu, S. M., Ibrahim, H. H., and Abdwahab, A. B. 2018. "Taxonomy development of blockchain platforms: Information systems perspectives". In: *AIP Conference Proceedings*. Vol. 2016. 1. AIP Publishing LLC, p. 020130.

Shin, S. I., Kim, J. B., Hall, D., and Lang, T. 2019. "What Information Propagates among the Public when an Initial Coin Offering (ICO) is Initiated? A theory-driven approach". In: *Proceedings of the 52nd Hawaii International Conference on System Sciences*.

Szabo, N. 1997. "Formalizing and securing relationships on public networks". *First monday* ().

Tasca, P. and Tessone, C. J. 2019. *A taxonomy of blockchain technologies: principles of identification and classification. Ledger 4 (2019)*.

Tönnissen, S. and Teuteberg, F. 2018. "Towards a taxonomy for smart contracts". In: *Twenty-Sixth European Conference on Information Systems*. Portsmouth.

Victor, F. and Lüders, B. K. 2019. "Measuring ethereum-based erc20 token networks". In: *International Conference on Financial Cryptography and Data Security*. Springer, pp. 113–129.

Wieninger, S., Schuh, G., and Fischer, V. 2019. "Development of a Blockchain Taxonomy". In: *2019 IEEE International Conference on Engineering, Technology and Innovation (ICE/ITMC)*. IEEE, pp. 1–9.

Wohrer, M. and Zdun, U. 2018. "Smart contracts: security patterns in the ethereum ecosystem and solidity". In: *2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*. IEEE, pp. 2–8.

Wöhrer, M. and Zdun, U. 2018. "Design patterns for smart contracts in the ethereum ecosystem". In: *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*. IEEE, pp. 1513–1520.

Xu, X., Pautasso, C., Zhu, L., Gramoli, V., Ponomarev, A., Tran, A. B., and Chen, S. 2016. "The blockchain as a software connector". In: *2016 13th Working IEEE/IFIP Conference on Software Architecture (WICSA)*. IEEE, pp. 182–191.

Yeow, K., Gani, A., Ahmad, R. W., Rodrigues, J. J., and Ko, K. 2017. "Decentralized consensus for edge-centric internet of things: A review, taxonomy, and research issues". *IEEE Access* (6), pp. 1513–1524.

Zheng, Z., Xie, S., Dai, H.-N., Chen, W., Chen, X., Weng, J., and Imran, M. 2020. "An overview on smart contracts: Challenges, advances and platforms". *Future Generation Computer Systems* (105), pp. 475–491.

Zou, W., Lo, D., Kochhar, P. S., Le, X.-B. D., Xia, X., Feng, Y., Chen, Z., and Xu, B. 2019. "Smart Contract Development: Challenges and Opportunities". *IEEE Transactions on Software Engineering* (), pp. 1–1.