# SoK: Development of Secure Smart Contracts – Lessons from a Graduate Course

Monika di Angelo[1,2(✉)], Christian Sack[1], and Gernot Salzer[1,2]

[1] Technische Universität Wien, Vienna, Austria
{monika.diangelo,christian.sack,gernot.salzer}@tuwien.ac.at
[2] Eurecom, Biot, France

**Abstract.** Smart contracts are programs on top of blockchains and cryptocurrencies. This new technology allows parties to exchange valuable assets without mutual trust, with smart contracts controlling the interaction between the parties. Developing smart contracts, or more generally decentralized applications, is challenging. First, they run in a concurrent environment that admits race conditions; adversaries may attack smart contracts by influencing the order of transactions. Second, the required functionality is often based on roles and states. This proves to be difficult to implement in current smart contract languages. Third, as a distinctive feature, smart contracts are immutable, hence bugs cannot be corrected easily. At the same time, bugs may cause (and have already caused) tremendous losses; they are to be avoided by all means.

This paper discusses our approach of *teaching* the development of secure smart contracts on the Ethereum platform at university level. This is a challenging task in many respects. The underlying technologies evolve rapidly and documentation lags behind. Available tools are in different stages of development, and even the most mature ones are still difficult to use. The development of secure smart contracts is not yet a well-established discipline. Our aim is to share our ideas, didactic concept, materials, insights, and lessons learned.

**Keywords:** Smart contract · Secure development · University course · Ethereum · Solidity

## 1 Introduction

Smart contracts were envisioned by Nick Szabo about 20 years ago [27,28] as computer programs automating the exchange of digital assets, which may be linked to non-digital objects or values. Smart contracts became effectively alive with the advent of cryptocurrencies. While playing only a limited role in Bitcoin, they are an essential ingredient of platforms like Ethereum [12].

Cryptocurrency-based smart contracts run on peer-to-peer networks that consist of mutually distrusting nodes (so-called miners) ideally operating in a decentralized manner. There is no need for an external trusted authority, miners execute smart contracts in an autonomous fashion.

### 1.1 Characteristics of Smart Contracts

The main characteristics of smart contracts are: *immutability* (as long as the community does not decide otherwise), *transparency* (when the ledger is open), provisioning of a *digital service* (or the digital mapping of a service), an *interface to the outside world* to enable interaction with it, *no central control/supervision* of transactions and contract execution, and for the contracting parties *no necessity to reveal their identities* to anyone.

Because of these properties smart contracts promise to be of use for applications requiring trustless computation, observability, tamper evidence, and decentralization. *Trustless computation* means autonomous execution of the program as well as no need for a trusted (third) party. Instead, trust is put in the ledger that keeps track of the exchange of assets. Smart contracts draw from the transparency technology of the underlying cryptocurrency, providing *observability and tamper evidence* as a trust base. *Decentralization* means that the system as a whole should not suffer from a single point of failure, and again should not rely on anything that needs to be trusted.

Application areas where such requirements may be desirable and useful are for example notary services, open government, insurance services, supply chain management, copyright management, and FinTech. Applications based on smart contracts are still in their infancy, with the most successful ones being initial coin offerings (ICOs) and collectibles like CryptoKitties [6].

### 1.2 Reasons for a University Course

Distributed applications (Dapps) use smart contract as backend to implement part of the business logic and to store critical data. Developing such Dapps is not just about learning another script language for smart contracts, but brings in considerable complexities that result e.g. from the concurrency, transparency, and immutability of transactions. Failing to acknowledge these complexities led to a situation where smart contracts are more famous for bugs and money losses than for success stories.

At the same time start-ups and traditional companies (like financial institutions and energy providers) jump on the bandwagon and urgently search for programmers able to develop Dapps that handle valuable assets reliably. This need is complemented by a massive interest of computer science students in this apparently hot topic. From a didactic point of view, smart contracts and blockchains are a worthwhile subject as they relate to many topics in computer science, like security, concurrency, cryptographic protocols, randomness, advanced algorithms, data structures, and formal verification.

### 1.3 Interesting Courses Online and at Other Universities

Among the available university based courses on Bitcoin, blockchains, and cryptocurrencies, the most prominent one is [19], a highly recommended book with a great accompanying online course.

For smart contracts in particular, information on held or available courses is scarce. The authors of [7] were the first to document the teaching of smart contracts as a university course. They report "several typical classes of mistakes [undergraduate] students made, suggest ways to fix/avoid them, and advocate best practices for programming smart contracts." The main problems were failure to encode the state machine properly, failure to use cryptography, misaligned incentives, and Ethereum specifics. This can be regarded as a reference course. For their lab they used Serpent, a high level programming language in the Ethereum world. Their pedagogical approach of "build, break, and amend your own program" seemed to be beneficial to teach adversarial thinking. As a conclusion about smart contracts, they arrived at "designing and implementing them correctly was a highly non-trivial task".

Outside academia and free to use on the internet, there are two projects which we consider well done and worthwhile. *Ethernaut* [20] is a war game, where vulnerabilities of smart contracts need to be exploited to advance. The instructive challenges have different levels of difficulty that are indicated accordingly. To succeed one has to understand known vulnerabilities and to apply the gained insights when using the provided tools. The online tutorial *CryptoZombies* [15] provides a nice gamification of how to develop smart contracts. It introduces the programming language Solidity in several steps, while jumping right into matters. By forming an army of zombies, one learns to program a suite of contracts similar to the popular CryptoKitties [6]. Finally, one is instructed on how to build a Dapp around the zombie contracts.

### 1.4 Added Value of This Paper

Teaching a subject like smart contracts that is new and in flux poses several challenges. There are no reference courses that may serve as a blue-print; the essence of secure smart contract programming has to be distilled from many sources. Moreover, the tools and techniques for the development of Dapps are still evolving and need to be evaluated regarding their suitability for teaching. When preparing the course, we came across a single report about a similar endeavor [7], which was helpful but at the same time already partly outdated.

The purpose of this report is to pass on our findings and experience in order to inspire and aid teachers designing similar courses as well as to identify difficulties encountered during the development of smart contracts. To this end, we present an analysis of the students' development efforts, a discussion of useful resources and tools, and a critical reflection of our experiences.

Furthermore, we contribute to defining the need for specific and qualifying university courses on blockchain and smart contracts programming. Based on the students' answers and feedback, we present insights into problems faced by researchers and developers when dealing with smart contracts. We elaborate on the issues encountered and provide several suggestions.

### 1.5 Roadmap

In the next section we present our course design in detail and the course setup. The lessons learned including an analysis of the students' development efforts is provided in Sect. 3. In Sect. 4 we discuss our approach, and in Sect. 5 we draw our conclusions about smart contract development and note further challenges.

## 2 Course Design

For the course design, we first define basic details like learning outcomes and content. Subsequently, we summarize the considered body of knowledge on which we base the learning activities. Then we present the assignments in detail.

### 2.1 Characteristics

**Learning Outcomes.** The aim of the course is that students gain knowledge and skills in the following areas.

*Technological foundations:* Understand the technological basis of smart contracts, like the blockchain, the Ethereum virtual machine (EVM), and the execution of smart contracts by miners.

*Programming languages and tools:* Use the languages, tools, and technologies for developing smart contracts and for interacting with them, like Solidity, Remix, Truffle, Geth, and Web3.js.

*Security and privacy issues:* Recognize and avoid security and privacy issues resulting either from the technology or from poor programming practices.

*Smart contracts and Dapps:* Develop secure smart contracts and Dapps involving tokens and cryptocurrencies.

**Course Contents.** In the lectures we recap the cryptographic basics as needed to understand cryptocurrencies and blockchain mechanics, explain the basic concepts of smart contracts, describe Ethereum in detail, highlight the peculiarities of scripting in Solidity, present the basics of the EVM as well as its relation to Solidity, and discuss current approaches to verification of smart contracts. The workshops cover tools and introductory exercises. The challenges address known vulnerabilities. Tokens and their usage as well as specific programming techniques are essential parts of the two projects.

**Target Audience.** Smart contracts and Dapps constitute advanced topics in computer science that presuppose knowledge in areas like algorithms, programming, web computing, and security. Therefore we devised *Smart Contracts* as an elective course in the master programmes of computer science and business informatics. Students without a background in Bitcoin and blockchain technology are referred to [19] and the accompanying video lectures.

### 2.2 Body of Knowledge

**Smart Contracts.** We start with Nick Szabo's ground breaking ideas on smart contracts [27,28] and Princeton's great introductory book and online course on cryptocurrencies [19]. The research perspectives and challenges for cryptocurrencies in [4] are worth considering, too. Regarding the Ethereum world, we refer to the Ethereum basics [12,31] and Buterin's blockchain and smart contract mechanism design challenges [29], as well as the overview of scripting languages in [24]. Furthermore, the article [26] with a legal perspective and the presentation [14] with a programming point of view offer additional perspectives on the topic. For platforms and use cases, [2] provides an interesting empirical analysis of smart contracts regarding platforms, applications, and design patterns, whereas [23] discusses decentralized applications. The challenges and new directions for blockchain-oriented software engineering in [22] provided useful insights, as did [17] with their elaboration on validation and verification of smart contracts.

**Security Issues.** [1] presents a useful survey of attacks on Ethereum smart contracts, whereas [16] not only investigates the security of smart contracts deployed on the Ethereum main chain, but also proposes to use symbolic execution (as implemented in the tool Oyente) to make contracts less vulnerable. [3] presents a declarative domain-specific language (Findel) to add security to financial agreements handled by smart contracts. The blog post [13] provides a guide to auditing smart contracts and reviews relevant attacks.

**Last Minute Contributions.** As smart contracts are a lively field, interesting work kept appearing throughout the course. This includes the collection of coding patterns [30] with proposals how to mitigate typical attacks. The authors of [8] encourage best practices to mitigate detrimental software behavior and argue for specific "Blockchain Software Engineering" since existing approaches seem insufficient for the particular needs of smart contract development.

### 2.3 Learning Activities

The course activities comprised lectures, workshops, and assignments, accompanied by a moderated discussion forum and email support.

**Lectures.** The main intention of the lectures was to cover new material deemed necessary to achieve the learning outcomes. While the usual course format is lectures with accompanying assignments, we deliberately added a workshop component.

**Workshops.** The intent of the workshops was to alleviate the frustration associated with using a range of new tools, and to close gaps in the understanding of the presented material. The students brought their own laptops to gain hands-on experience. We first demonstrated the handling of selected tools and assisted with initial problems arising from the partly unstable tool chain and the incomplete documentation. Then, we focused on small ad-hoc tasks to make sure everyone was familiar with the operations and concepts required for the upcoming assignments. The workshops also served the purpose of discussing sample solutions after the submission deadlines.

**Assignments.** The assignments were intended to provide students with practical experience regarding the implementation of smart contract, to let the students apply the newly gained skills and knowledge, and for us to get feedback on the progress of students regarding their understanding of the essential concepts.

The assignments started with the online tutorial *CryptoZombies* [15] (see Sect. 1.3) in order to provide an entertaining introduction into programming in Solidity. Subsequently, eight security challenges had to be solved, with the intention to get the students to understand known vulnerabilities and to motivate the need for secure smart contract development.

Finally, there were two constructive tasks. For the guided project "beer bar" we provided a clear specification and an abstract Solidity contract including comments for the parts to be implemented by the students. The final project had a free topic and just a few constraints.

Ether was only available in limited quantities to raise awareness that it is a costly resource. Next to a regular supply of Ether which was sufficient to solve the assignments, it was also handed out as a reward for participation in the workshop tasks, and upon request.

## 2.4 Assignments in Detail

**Challenges.** The eight security challenges (inspired by Ethernaut [20]) are packed into a story in which the main character is a software developer. For each challenge the task is to deplete the provided contract by finding and successfully exploiting one or more security issues. The challenges address known vulnerabilities concerning the fallback function, a misnamed constructor, math issues like overflow, forced transfer of Ether, reentrancy, hidden variables, delegatecall, insecure contract interaction, failing transactions, and randomness.

**Beer Bar.** This assignment consists of four constructive sub-tasks.

*Task 1.* The students implement a *bar token* that is not divisible, but mintable and burnable. Furthermore, they make sure that tokens cannot get lost by sending them accidentally to contracts that are not set up for accepting them. The concept of tokens as well as standard token contracts [21] were introduced in the workshop. To solve this task the students customize an ERC223 token.

*Task 2.* The students implement a *beer bar* that uses the bar token from Task 1. We provide the interface as well as a skeleton contract with in-line comments that describe the functionality to be implemented, like opening and closing the bar, setting the beer price and the bar token to be accepted, and the processing of beer orders. For modeling the roles of bar owners and bar keepers, the students use the `RBAC` contract [21].

*Task 3.* The students extend the bar to a *song voting bar* where customers can vote for the songs to be added to the playlist, with an additional role `DJ`.

*Task 4.* We provide a simple web interface in Javascript that uses `web3.js` to communicate with the contract of the beer bar. The students extend the interface to interact with their song voting bar.

**Final Project.** For the final project, the students are encouraged to choose a topic of their own. If lacking inspiration, they may extend the beer bar. The final project is graded with respect to the following criteria.

- Use of mappings, `RBAC`-roles, modifiers, Ether, tokens, correct math.
- Some (minimalistic) web interface for interacting with the contract.
- The contracts should not make any Ether or tokens inaccessible.
- The contracts should not exhibit any of the vulnerabilities discussed in the security challenges before.
- Quality of the documentation specifying the contract and the web interface.

The following aspects give an additional bonus: original choice of topic, commitment schemes for guarding secrets (like bids or game moves), deposits to prevent aborts or reverts of games, timeouts to ensure the termination of moves or games, and good randomness.

### 2.5 Technical Setup

The technical setup of the course consists of a Linux server providing an Ethereum blockchain and a block explorer, as well as a separate client for the lecturers and the students (Linux, MacOS, and Windows). As implementation languages we use *Bash, Javascript, Solidity,* and *Html.*

**Ethereum Blockchain.** Geth [11] runs a private chain with proof of authority (POA). The geth client (miner) has to be configured such that it can handle sufficiently many concurrent connections to give all students access in parallel. Moreover, it turned out that we need a high block gas limit for publishing the challenges (see below).

**Block Explorer.** We developed our own block explorer consisting of a single Html page and some programs in Javascript. When the user opens the block explorer in a browser it connects to the local geth instance to load the chain data. As a result, the user sees the local synchronized state of the blockchain, reducing the network load on our server. A filter allows the students to restrict their view to the transactions they are actually involved in. This helps in situations where several students are active at the same time (like during workshops).

**Administration of Students and Assignments via the Blockchain.** We deployed several contracts on our blockchain to manage the submission of assignments and the interaction with and between students.

The *address book* provides a unique mapping between student ids and Ethereum addresses (one *public* address per student); moreover, it maintains a list of several *private* Ethereum addresses per student. The public address is used e.g. for interactions between students and for transferring Ether.

The addresses of personalized copies of challenges and of contracts submitted by a student are stored as private addresses, only known to a single student and to the lecturers.

The *Ether tap* regularly transfers small amounts of Ether to all public addresses. Moreover, in case of mishaps (like transferring accidentally all Ether to the zero address) students may request limited amounts of Ether from this contract.

The *alias directory* allows the students to specify a string to be used in place of their name. This alias is later used for displaying live progress visualizations and feedback (e.g. who has already solved a task) and rankings (like the time needed to solve a challenge).

A *base contract* is inherited by each personalized copy of a challenge. It adds private variables that ensure that the students can only interact with their own copy, and that the challenge can be turned off after the deadline.

**Client for Lecturers.** The client for lecturers is a geth client with console scripts attached to it. It automatically prints each transaction as it occurs, improving readability by translating the involved addresses to names. The scripts provide functions for easy maintenance and observation during the course. As an example, the function `balances` identifies students who have spent most of their Ether. Other scripts deploy a challenge for a single student or groups of addresses, whereby certain values in each instance can be varied randomly to personalize the challenge for each student.

**Client for Students.** The client for the students is a geth client connecting automatically to the private course chain. It includes the abstract binary interface (ABI) and the deployment address of the address book such that students can access their private and public addresses in a symbolic way.

## 3 Lessons Learned from the Students' Submissions

### 3.1 Beer Bar

It turned out to be difficult for some students (25%) to accept only their own tokens (and not arbitrary ones). After the submission deadline, we addressed this issue in the workshop. We reopened the submission for the beer bar to admit corrections, because we felt this token aspect was too important to miss.

### 3.2 Final Project

After teaching known vulnerabilities by means of security challenges to motivate secure smart contracts, and after showing best practice examples, we were interested to see which topics the students would choose for their final project and how they would implement them.

The final project was handed in by 44 students. Most students addressed the *homo ludens* by implementing some sort of game (14), gambling (13), betting (5), or lottery (3). Three students opted for a shop. The remaining six students came up with extra-ordinary topics, namely *cash register functionality that conforms to local law*, *untraceable and unlinkable voting (using a ring signature)*, *smart marketplace*, *data saver*, *trusted recommender*, and *betterSpotify*.

The effort put into the final project varied. 14 students pragmatically extended their implementation of the beer bar or transformed it into a shop of similar structure. Several students skipped the web interface (10) or documented their project poorly (6). At the other end, about half of the students went at great lengths to provide a rounded-off project, paying also attention to details.

The students encountered several difficulties in their implementation efforts. Our insights are broken down with respect to the requirements for the project.

**Roles, Modifiers, Require Statements.** These elements posed no problems. They are mentioned first in the initial tutorial on Solidity, CrytoZombies, and then are repeatedly used throughout the challenges and in the guided project.

**Data Structures.** Many students wanted to iterate over arrays. We explained during the course that loops over potentially growing ranges should be avoided as the execution may eventually exceed the gas limit and fail (apart from becoming more and more expensive). Not being able to iterate over mappings required time to get used to. Deciding which part of the data and the program logic should be put on-chain and which one off-chain remained an issue.

**Correct Math.** It was easy for the students to understand the problems of overflows and wrap-arounds and to take precautions.

**Token Usage.** Even though we put an emphasis on tokens and their correct usage, some students had troubles. This was particularly true for ERC223 tokens and the idea of the token fallback function. These concepts presumably need more time, explanations, and exercises.

**Commitments.** The correct usage of commitments needs basic security knowledge that we did not teach, just briefly summarized. Most students did not require commitments for their final project. Several students (13) employed them correctly, a few tried but failed.

**Stages/Phases.** The usage of stages or phases in order to ensure the correct ordering of transactions did not pose any problems.

**Randomness.** Good randomness within smart contracts is generally tricky. We only covered it superficially. Most projects did not require (good) randomness. Some students made serious attempts, most of them (9) succeeded.

**Private Variables.** Even though we covered private variables in the workshops and the challenges, quite a few students had problems with them. The keyword *private* and the missing getter function seemed to make them blind to the fact that the variables could still be inspected from the outside. Apparently, private variables represent new and unexpected material that needs more exercises.

It might also be worthwhile to modify the syntax of Solidity by replacing the misleading keyword by another one, maybe *local* or *internal*.

**Deposits/Timeouts.** The usage of deposits or timeouts to handle unfair game aborts or stalling did not pose any problems.

**Web User Interface.** Students with little prior knowledge of web scripting had a difficult time implementing a basic web interface for their contracts, even though we provided an exemplary one for the bar contract. If this is to be an integral part of the course, it definitely needs more coverage or prior knowledge.

### 3.3 Tools

All students initially used Remix, and most of them stayed with this browser IDE. Some additionally tried Remixd to access the local file system; this add-on, however, did not prove stable and lead to the destruction of files in two cases.

Some students switched to the Truffle framework because of its promise of more structured testing and better handling of multi-contract projects. Testing is generally an issue with little or unstable support by the tools. A problem with Truffle was that a new version became available during the course that was not fully compatible with the old one.

## 4 Discussion

The choice of Ethereum as a platform for smart contracts was determined by the wealth of available materials that is unparalleled compared to any alternative platform.

### 4.1 Distinctive Aspects

Our approach differs from [7] in the following aspects.

**Focus.** We teach how to develop secure smart contracts. In their final assignment the students were asked to implement a project of their choice, after experiencing lectures, workshops, security challenges, and a guided project.

**Technological Environment.** There is more practical knowledge on smart contracts now. Security issues related to the programming language shifted. The tool chain is improving, but still leaves much to be desired.

**Didactic Design.** We worked with graduate students, security challenges based on known vulnerabilities, workshops with ad hoc tasks and live feedback, a guided project using tokens, and a final project with an open topic.

### 4.2 Course Feedback

The university provides a non-obligatory course questionnaire for the students to fill in at the end of each term. In this questionnaire, students are asked to answer a bit over 20 questions, rating their satisfaction from 1 (very content) to 5 (not at all content). Questions concern the preparation, implementation, interaction, and knowledge gain of a course.

16 out of the initially 53 students took the opportunity to give a feedback on our course. This is a high percentage (30%) compared to the usual less than 10%. Our course yielded an average graduation between 1.0 and 1.56 on the questions with a median value of 1. In comparison, the typical median for courses of the faculty is 2. These numbers indicate a high satisfaction with the course.

Verbal feedback included statements like: "One of the best courses I've attended so far." "Previously, I had about 0 interest and prior knowledge of blockchain and cryptocurrencies and just attended the course to learn more. It definitely caught my interest." "Knowledge growth is still understated. It has opened a gateway to a new world!" "Really good and above all entertaining course."

Students especially liked: "The workshops and the course chain. That everything happens on the chain is pretty cool :) The exercises were always fun and well prepared." "The whole course was really great. Please do a continuation course." "Workshops, panel discussion" "Security challenges, course chain setup, general format with course + workshop" "Challenges were nicely designed. Also given creative freedom, because you can make the final project completely yourself." "The workshops were absolutely great. Originally I did not want to attend because of time constraints, but they were just too good to omit (100% attendance). The challenges were a lot of fun and were just at the right level of difficulty. 1–2 a bit too heavy for me on my own, but with tiny hints from others they were no problem." "The format with 1 hour lecture + 2 hours workshops + the plenary discussion in the last lecture. The course environment with its own chain is great."

Potential for improvement was seen in: "The pace was a bit high." "The effort was too high, even if it's fun ..." "The web part needs more introduction and support." "More time for the final project." "A recording of the lectures would be helpful." "I would have liked a model solution of the guided project."

## 5 Conclusions

Smart contracts are an interesting topic to teach in computer science, since they combine areas like distributed systems, security, data structures, software engineering, algorithms, and formal verification. There are connections to finance (values at stake) and law (legal aspects of the usage of smart contracts).

Secure smart contracts are still avant-garde. Even though there are coding patterns and best practice collections for most known (security) bugs (like [21]), the development of secure smart contracts is not yet a well-established discipline.

### 5.1 Differences to Conventional Development

Summarizing our experience from the course on smart contracts, we identified the following issues in developing secure smart contracts. We started to address them and raised awareness in these regards. For smart contracts to become a reliable technology, these issues should be addressed further.

**Security is an Issue.** Developing secure smart contracts needs 'adversarial thinking'. A key feature of smart contracts is their immutability once deployed on the blockchain, because this feature is part of the trust base. So, smart contracts have to be designed and implemented correctly right from the start with little chances for updates. Even though an update strategy is possible, it deteriorates the trust base. Moreover, smart contracts usually have values at stake. As they are intended to work autonomously once they are released, there is a non-negligible incentive for adversaries to exploit potential vulnerabilities.

**Underspecification is an Issue.** In general, it is difficult for developers to consider all possible program states and transitions and to fully specify the behavior of the program, especially if there is little or no tool support for it. A full specification (with the aid of a suitable tool) seems still a long way to go, but would be a prerequisite for the verification of smart contracts. There are currently only few approaches [5,13,16,18] that help developers find overlooked states. This goes hand in hand with security issues as underspecification readily leads to a vulnerability. Again, there is an economic incentive to exploit potential gaps, and the immutability requires to close the gaps beforehand. Maybe a game theoretic approach would help to explicitly balance the incentives that smart contracts (implicitly) create.

**Concurrency is an Issue.** Smart contracts are decentralized pieces of trustless computer code. Although they essentially run on the hardware of a miner, and some advocate considering the miner network as a large linear 'world computer', there are some concurrent aspects in smart contracts. Some program sequences require more than one transaction. Even though a single transaction is an atomic operation, multiple transactions cannot be bundled to a single atomic operation. Once a transaction is on the blockchain, it cannot be reversed by a following transaction. There can be race conditions, transaction ordering makes a difference. It can be influenced (by transaction fees), but not relied upon within a smart contract (without special handling like stages). Understanding the concurrency aspects of smart contracts is still a research topic [9,25].

**Novelty is an Issue.** Blockchains and cryptocurrencies are still evolving, and so are the programming languages for smart contracts (such as Solidity). Tools in this domain are developed for a moving target, and thus barely develop beyond beta status before becoming obsolete. Best practice and coding patterns are gradually emerging (e.g. OpenZeppelin [21]). Working with new technologies is quite a challenge, even more when they keep changing.

**Separation is an Issue.** Some of the involved data and logic have to stay on the chain (via transactions) in order to maintain their security and integrity, while others do not need the costly features of a blockchain. All parts with no need to be handled on-chain should be handled off-chain, in order to reduce (transaction) costs and execution time as well as capacity issues. It may be necessary to provide validation (in terms of integrity and security) for some of the off-chain processes and data (e.g. authenticity proofs). E.g. [10] discusses such off-chain patterns. Also, a static (or perhaps even dynamic) analysis is required to decide which pieces of data and logic are best handled on-chain.

## 5.2 Further Challenges

For the development of secure smart contracts there are several areas with potential for improvement.

*Platforms.* Besides Ethereum, there are current and planned platforms, which intend to solve several issues (e.g. consensus, performance). It will remain interesting to observe, evaluate, and contribute to ongoing developments that provide a suitable basis for secure smart contracts.

*Scalability.* The number of transactions per second is still an open issue, as well as the growing size of the chain. Moreover, the execution speed of smart contracts may constitute a bottleneck.

*Development Frameworks.* Currently there are a few tools with basic development support. Even though they are rapidly evolving, there is still insufficient support for secure implementations. Especially, support for correct and complete specifications would be of great help.

*Programming Languages.* The currently prevalent language Solidity is still evolving with incomplete documentation lagging behind. Other languages exist or are being developed. Again, support for correct and complete specifications would be of great help. Programming languages for smart contracts are an interesting field of ongoing research.

*Verification.* Because of the high value at stake paired with the immutability of deployed code, (formal) verification of smart contracts is desirable. This is also an interesting area of ongoing research.

**Acknowledgments.** We are grateful to our students who made the course a worthwhile experience with their enthusiastic participation and high quality assignments. We would like to extend our sincere thanks to our student assistants for their commitment and dedication in the preparation phase as well as during the course. A special thanks goes to our two guest lecturers from the Vienna based security research company SBA.

## References

1. Atzei, N., Bartoletti, M., Cimoli, T.: A survey of attacks on Ethereum smart contracts (SoK). In: Maffei, M., Ryan, M. (eds.) POST 2017. LNCS, vol. 10204, pp. 164–186. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54455-6_8
2. Bartoletti, M., Pompianu, L.: An empirical analysis of smart contracts: platforms, applications, and design patterns. In: Brenner, M., et al. (eds.) FC 2017. LNCS, vol. 10323, pp. 494–509. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-70278-0_31
3. Biryukov, A., Khovratovich, D., Tikhomirov, S.: Findel: secure derivative contracts for ethereum. In: Brenner, M., et al. (eds.) FC 2017. LNCS, vol. 10323, pp. 453–467. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-70278-0_28
4. Bonneau, J., Miller, A., Clark, J., Narayanan, A., Kroll, J.A., Felten, E.W.: SoK: research perspectives and challenges for bitcoin and cryptocurrencies. In: IEEE Symposium on Security and Privacy (SP 2015), pp. 104–121. IEEE Computer Society (2015). https://doi.org/10.1109/SP.2015.14
5. Bragagnolo, S., Rocha, H., Denker, M., Ducasse, S.: SmartInspect: smart contract inspection Technical report. Ph.D. thesis, Inria Lille (2017). https://hal.inria.fr/hal-01671196/document
6. Dapper Labs Inc: CryptoKitties. https://www.cryptokitties.co. Accessed 07 Aug 2018
7. Delmolino, K., Arnett, M., Kosba, A., Miller, A., Shi, E.: Step by step towards creating a safe smart contract: lessons and insights from a cryptocurrency lab. In: Clark, J., Meiklejohn, S., Ryan, P.Y.A., Wallach, D., Brenner, M., Rohloff, K. (eds.) FC 2016. LNCS, vol. 9604, pp. 79–94. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-53357-4_6
8. Destefanis, G., Marchesi, M., Ortu, M., Tonelli, R., Bracciali, A., Hierons, R.M.: Smart contracts vulnerabilities: a call for blockchain software engineering? In: 2018 International Workshop on Blockchain Oriented Software Engineering, pp. 19–25. IEEE Computer Society (2018). https://doi.org/10.1109/IWBOSE.2018.8327567
9. Dickerson, T., Gazzillo, P., Herlihy, M., Koskinen, E.: Adding concurrency to smart contracts. In: ACM Symposium on Principles of Distributed Computing (PODC 2017), pp. 303–312. ACM, New York (2017). https://doi.org/10.1145/3087801.3087835
10. Eberhardt, J., Tai, S.: On or off the blockchain? Insights on off-chaining computation and data. In: De Paoli, F., Schulte, S., Broch Johnsen, E. (eds.) ESOCC 2017. LNCS, vol. 10465, pp. 3–15. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-67262-5_1
11. Ethereum Foundation: Go Ethereum - the Ethereum protocol implemented in Go. https://geth.ethereum.org. Accessed 11 Sept 2018
12. Ethereum Wiki: A next-generation smart contract and decentralized application platform. https://github.com/ethereum/wiki/wiki/White-Paper. Accessed 29 July 2018
13. Grincalaitis, M.: The ultimate guide to audit a smart contract and the most dangerous attacks in Solidity (2017). https://medium.com/@merunasgrincalaitis/how-to-audit-a-smart-contract-most-dangerous-attacks-in-solidity-ae402a7e7868. Accessed 09 Aug 2018
14. Henglein, F.: Smart contracts are neither smart nor contracts (slides) (2017). http://hjemmesider.diku.dk/~henglein/smart-contracts-are-neither.pdf. Accessed 09 Aug 2018

15. Loom Network: CryptoZombies. https://cryptozombies.io. Accessed 07 Aug 2018
16. Luu, L., Chu, D., Olickel, H., Saxena, P., Hobor, A.: Making smart contracts smarter. In: Weippl, E.R., et al. (ed.) 2016 ACM SIGSAC Conference on Computer and Communications Security, pp. 254–269. ACM (2016). https://doi.org/10.1145/2976749.2978309
17. Magazzeni, D., McBurney, P., Nash, W.: Validation and verification of smart contracts: a research agenda. IEEE Comput. **50**(9), 50–57 (2017). https://doi.org/10.1109/MC.2017.3571045
18. Mavridou, A., Laszka, A.: Tool demonstration: FSolidM for designing secure ethereum smart contracts. In: Bauer, L., Küsters, R. (eds.) POST 2018. LNCS, vol. 10804, pp. 270–277. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89722-6_11
19. Narayanan, A., Bonneau, J., Felten, E., Miller, A., Goldfeder, S.: Bitcoin and Cryptocurrency Technologies: A Comprehensive Introduction. Princeton University Press, Princeton (2016)
20. OpenZeppelin: Ethernaut - Solidity security challenges. https://github.com/OpenZeppelin/ethernaut. Accessed 07 Aug 2018
21. OpenZeppelin: Solidity contract library. https://github.com/OpenZeppelin/openzeppelin-solidity. Accessed 07 Aug 2018
22. Porru, S., Pinna, A., Marchesi, M., Tonelli, R.: Blockchain-oriented software engineering: challenges and new directions. In: Uchitel, S., et al. (ed.) 39th International Conference on Software Engineering (ICSE 2017), pp. 169–171. IEEE Computer Society (2017). https://doi.org/10.1109/ICSE-C.2017.142
23. Raval, S.: Decentralized Applications: Harnessing Bitcoin's Blockchain Technology. O'Reilly Media, Newton (2016)
24. Seijas, P.L., Thompson, S.J., McAdams, D.: Scripting smart contracts for distributed ledger technology. IACR Cryptol. ePrint Archive 2016/1156 (2016). http://eprint.iacr.org/2016/1156
25. Sergey, I., Hobor, A.: A concurrent perspective on smart contracts. In: Brenner, M., et al. (eds.) FC 2017. LNCS, vol. 10323, pp. 478–493. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-70278-0_30
26. Sreehari, P., Nandakishore, M., Krishna, G., Jacob, J., Shibu, V.S.: Smart will converting the legal testament into a smart contract. In: 2017 International Conference on Networks Advances in Computational Technologies (NetACT), pp. 203–207, July 2017. https://doi.org/10.1109/NETACT.2017.8076767
27. Szabo, N.: Formalizing and securing relationships on public networks. First Monday **2**(9), 28 (1997). https://doi.org/10.5210/fm.v2i9.548
28. Szabo, N.: Secure Property Titles with Owner Authority (1998). http://nakamotoinstitute.org/secure-property-titles/. Accessed 09 Aug 2018
29. Vitalik, B.: Blockchain and smart contract mechanism design challenges (slides) (2017). http://fc17.ifca.ai/wtsc/Vitalik%20Malta.pdf. Accessed 09 Aug 2018
30. Wöhrer, M., Zdun, U.: Smart contracts: security patterns in the Ethereum ecosystem and Solidity. In: 2018 International Workshop on Blockchain Oriented Software Engineering, pp. 2–8. IEEE Computer Society (2018). https://doi.org/10.1109/IWBOSE.2018.8327565
31. Wood, G.: Ethereum: a secure decentralised generalised transaction ledger. Technical report, Ethereum Project Yellow Paper (2014). https://ethereum.github.io/yellowpaper/paper.pdf. Accessed 09 Aug 2018