CS 4973/6983 Smart Contracts and Analysis – Fall 2025 Lecture Notes UNDER CONSTRUCTION

Stavros Tripakis
https://www.ccs.neu.edu/~stavros/

September 8, 2025

Abstract

This course covers topics in program analysis, focusing on smart contracts. We explore the world of blockchains, smart contracts, and decentralized finance (DeFi), focusing on Solidity programs running on Ethereum, and using tools such as Remix and Foundry. We also explore formal modeling and verification techniques (model checking, inductive invariants, and more) using tools such as TLA+, Spin, Z3, Certora, Dafny, and Lean. We also touch upon related topics such as cryptography and zero knowledge proofs, and decentralized/digital democracy.

This course is a seminar and requires active participation from students. The course is open to advanced undergraduates and graduate students. Students are expected to read and write code, papers, online material, etc. They are also expected to present such material and lead classroom discussions.

No prerequisites.

Contents

1 Introduction

1.1 Cryptocurrencies: money without banks

In 2008, so-called Satoshi Nakamoto published the white paper *Bitcoin: A Peer-to-Peer Electronic Cash System*. The paper, which is available online – https://bitcoin.org/bitcoin.pdf – can be considered to be the birth of *cryptocurrencies*. Satoshi Nakamoto is a pseudonym. To this date, the identity of the author(s) of the Bitcoin white paper is still unknown. To make our lives easier, we will refer to Satoshi Nakamoto as to a real person. Even though the real identity of Nakamoto is unknown, the motivation behind Bitcoin is clear. The first sentence in Nakamoto's paper reads:

> "A purely peer-to-peer version of electronic cash would allow online payments to be sent directly from one party to another without going through a financial institution."

In other words, the motivation is to create a *purely peer-to-peer payment system without the need of a financial institution*. In short, to create *money without banks*.

1.2 Blockchains: decentralized trust

The Bitcoin white paper can be considered to be not only the birth of cryptocurrencies, but also the birth of *blockchains*.[1] A blockchain is a *distributed protocol* that solves the problem of *distributed consensus*. What it really solves is the problem of *trust*. Think about it: if you want to create money without a bank, you need to solve the problem of trust. If you create your own "coin" and try to use it for payments, you need to convince people to accept it. You tell them that your coin has value, but why should they trust you? Why do people trust that the dollar has value? Because dollars are issued by banks, which are supposed to be trustworth authorities. But how can we create trustworthy money without a bank? Quoting again from the Bitcoin white paper:

> "What is needed is an electronic payment system based on cryptographic proof instead of trust, allowing any two willing parties to transact directly with each other without the need for a trusted third party."

The notion of trust is somewhat abstract. A more concrete notion is that of *agreement*. We all agree that pigs cannot fly. We all agree that the sky is blue. We all agree that the dollar has value.

In computer science, the notion of agreement has been formalized by the notion of *consensus*. Consensus is one of the most fundamental problems in computer science, and in particular in distributed systems. The consensus problem is to have a set of distributed *nodes* agree on something. The nodes (also called *agents* or *processes*) can simply be a set of computers, e.g., the computers in a data center, or all the computers connected to the internet.

The consensus problem arises in distributed systems all the time. For example, say you are a bank and you have a database that holds all your customers' accounts. The database holds entries that say things like *Mary's account has balance $1,234*, *Stavros' account has balance $1,000,000* (I wish!), etc. For reasons of fault tolerance, the database is *decentralized*, meaning it's not stored in one computer, but in many (so that if one computer breaks the data is not lost). But now that the database is distributed over many machines, the problem of consensus arises: all the machines must agree on all the balances at all times! Is that possible, and how? Amazingly, it turns out that blockchains make this possible: we will see how in §5.

1.3 Smart Contracts: Decentralized Finance (and more)

In 2014, Vitalik Buterin pushed the ideas of Bitcoin one step further in his white paper *Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform*. The Ethereum paper, which is available

[1] Even though the word *blockchain* does not appear in Nakamoto's white paper, what is described is indeed a chain of blocks, i.e., a blockchain.

online – https://ethereum.org/en/whitepaper/ – can be considered to be the birth of *smart contracts*.[2] The idea behind smart contracts is to use the blockchain as a *platform* on top of which many applications can be built (just like IP is the platform on top of which all sorts of internet applications are built). What kind of applications would one want in addition to Bitcoin? Many. For example: Bitcoin is just one cryptocurrency; can we let people create their own cryptocurrencies? Another example is a *smart property* ledger: can we use the blockchain to record who owns what property? Yet another example is a *decentralized market*: can we use the blockchain to create something like a stock exchange, but without a centralized trusted authority?

Bitcoin has some capabilities for creating such applications (e.g., see https://en.bitcoin.it/wiki/Contract). However, these capabilities are limited: one can say that smart contracts are not *first-class citizens* in Bitcoin. The Ethereum blockchain, with its programming language Solidity, aim to fill this gap. To quote from Buterin's 2014 paper:

> "Commonly cited applications include using on-blockchain digital assets to represent custom currencies and financial instruments ("colored coins"), the ownership of an underlying physical device ("smart property"), non-fungible assets such as domain names ("Namecoin") as well as more advanced applications such as decentralized exchange, financial derivatives, peer-to-peer gambling and on-blockchain identity and reputation systems. Another important area of inquiry is "smart contracts" - systems which automatically move digital assets according to arbitrary pre-specified rules. For example, one might have a treasury contract of the form "A can withdraw up to X currency units per day, B can withdraw up to Y per day, A and B together can withdraw anything, and A can shut off B's ability to withdraw". The logical extension of this is decentralized autonomous organizations (DAOs) - long-term smart contracts that contain the assets and encode the bylaws of an entire organization. What Ethereum intends to provide is a blockchain with a built-in fully fledged Turing-complete programming language that can be used to create "contracts" that can be used to encode arbitrary state transition functions, allowing users to create any of the systems described above, as well as many others that we have not yet imagined, simply by writing up the logic in a few lines of code."

According to https://etherscan.io/charts there are over 80 million smart contracts deployed on Ethereum at the time of writing. Etherscan also maintains an Ethereum Daily Deployed Contracts Chart – https://etherscan.io/chart/deployed-contracts – which shows thousands (and sometimes hundreds of thousands) of contracts being deployed every day. It is worth noting that not all deployed contracts are "useful". Some may no longer be "live", meaning they are no longer being "called" (we will learn what that means). The analysis in [42] found that 70% of the contracts deployed in a certain year were never called. ♠
Also, the live contracts may not necessarily be all different from each other. The page https://bitkan.com/learn/how-many-smart-contracts-on-ethereum-how-do-ethereum-smart-contracts-work-8989 states that "of the 15 million or so live contracts, about 70 percent are copies of one of the 15 templates." These statistics still leave a large number of active and interesting contracts.[3]

Ethereum daily transactions number in the millions – https://etherscan.io/chart/tx. There are over a million and a half *tokens* [4] with market capitalizations in the billions of dollars – https://etherscan.io/tokens. Note that Bitcoin and Ethereum are not the only blockchains out there (there are many more). ♠
Also note that Ethereum is not the only blockchain on which one can write and deploy smart contracts.

Smart contracts collectively enable and implement what is called *Decentralized Finance* or DeFi. According to https://www.grandviewresearch.com/industry-analysis/decentralized-finance-market-report "the global DeFi market size was estimated at USD 20.48 billion in 2024 and is projected to reach USD 231.19 billion by 2030." All the numbers are to be taken with a grain of salt. They are not given here ♠

[2]The Ethereum paper also credits Nick Szabo for describing some of the concepts behind smart contracts earlier.

[3]The ♠ at the margin indicates an invitation for students to add/refine/correct the corresponding content. I will be adding such invitations throughout the document. Students should also feel free to indicate places where the text should be refined or corrected, or where more info should be added.

[4]A *token* here refers to an ERC20 token. We will learn what that is later in this course. For now, you can think of a token as corresponding to a digital currency. For example, some tokens listed in https://etherscan.io/tokens are USDT, BNB, USDC, WBTC, WETH, and so on.

to impress you. Although DeFi is an important financial activity, it represents a very small percentage of global, traditional financial activity. It is unclear at this point whether this will ever change. Will DeFi become the norm in the future, or will it remain somewhat niche, marginalized? I don't know. The main reason to study smart contracts is not because there's a lot of money involved. The main reasons to student smart contracts, and by extension the main reasons for this course, are intellectual. Blockchains and smart contracts are a very interesting domain of computer science and beyond, and offer many opportunities to expand your horizons in terms of education.

I also want to emphasize at this point that this course *should by no means be taken as offering financial advice of any sort*. I am *not* a financial advisor. I am a university professor and my goal is to educate my students, and myself in the process. The goal is education, not profit.

DeFi applications are not the only applications of smart contracts, e.g., see `https://soliditylang.org/use-cases/`. This class focuses on DeFi, but we will also discuss some other applications, c.f. §1.7 and §8.

1.4 Security problems and attacks

Smart contracts are software. Like any other software, smart contracts have bugs and security vulnerabilities. Because smart contracts handle money, these bugs and vulnerabilities can result in losing money and sometimes very large amounts of money. Indeed, smart contracts are routinely scrutinized by security experts using code *audits* and other methods, some of which we will cover in this course. Despite these efforts, smart contracts are often attacked. Some of the most notorious attacks are listed below: ♠

- The DAO attack which occurred on June 17, 2016, and allowed the attacker to steal around $50 million worth of the virtual currency ETH (ether), or one third of the contract's total assets at the time. We discuss this attack in detail in §3.4.1.
- The First Parity Wallet Hack which occurred on July 19, 2017, and allowed the attacker to steal, according to some sources, over 150,000 ETH (≈30M USD at the time) [43, 20, 58].
- The Second Parity Wallet Hack which occurred on November 6, 2017, and allowed the attacker to "destroy", according to some sources, another 513,774.16 ETH, then valued at over $150 million [43, 20, 58]. We discuss both Parity Wallet attacks in detail in §3.4.2.

You might think that these attacks are historic: they happened a long time ago, "people learned their lessons" so to speak, and the problem has gone away. But this is not true. Attacks keep happening all the time, and keep security experts busy. For example, here are some blogs detailing recent attacks at the time of writing:

- The CPIMP attack which happened in July 2025 [50].
- The $11M Cork Protocol Hack which happened on the 28th of May 2025 [16].
- The Cetus AMM $200M Hack which happened on May 22, 2025 [14].
- The Bedrock vulnerability which was discovered on 26 September 2024 [15].

1.5 Mitigations – Audits, War Rooms, and Formal Methods

So, there are many smart contracts out there, holding a lot of money, and they are vulnerable and subject to continuous attacks. How do we mitigate this situation? This is not simple, because there are many things that need to be done:

- There must be ways to *detect* attacks and *alert* the stakeholders that an attack is happening (or has happened already).

- Once an attack is detected, there must be ways to defend against the attack as quickly as possible (ideally, stop the attack).
- Similarly, there must be ways to detect *vulnerabilities* to already deployed contracts (independently of whether these vulnerabilities have already been exploited into attacks) and defend against such vulnerabilities as quickly as possible, to avoid them materializing into attacks.
- There must be ways to discover vulnerabilities *prior to deploying* a contract.

There are more formal and less formal ways to go about addressing the issues above. Among the less formal are *code audits*, which involve carefully reading the source code of the smart contract prior to deployment, with the intent of discovering bugs and vulnerabilities. However, code audits may very well go beyond just looking at the code ("eye-balling") and involve more formal techniques, such as running different kind of tools (e.g., static analysis or verification) on the code, or deploying and executing/testing the code itself on sandboxes. In the end, the company tasked with performing the audit (which is typically different from the company that developed the code in the first place) issues an *audit report*. Several such reports are available online, e.g. [40, 48].

Of course, the need to look at the code carefully may arise even after the smart contract is deployed. For instance, in the event of an attack, or when a vulnerability is discovered, security experts often assemble so-called *war rooms* which try to find solutions and take actions to prevent attacks or stop them while they are happening, as well as to alert stakeholders, e.g., see [50, 15].

This course focuses more on the *formal* side of techniques. You will gain an understanding and appreciation of the term *formal* throughout the course. For now, let's just say that formal techniques strive to be rigorous, mathematically grounded, and hence to provide stronger guarantees (more discussion in §4.1). There are many many formal techniques out there, enough to fill many many university courses (see §4.1 for some courses that I teach on formal techniques). In this course, we will focus on so-called *formal methods* and in particular *formal verification*. We will discuss these extensively throughout the course.

The goal of formal verification is to *prove* that a piece of software works correctly *under all circumstances*. For example, if we are talking about a simple piece of software, e.g., a single method, we may want to guarantee that the method outputs the correct result for *any* input. We typically cannot do this simply by testing, because with testing we only run the program on a finite number of inputs, typically much smaller than the set of all possible inputs. In fact, the set of inputs can a-priori be infinite. Is it possible in that case to guarantee correctness without running an infinite number of tests? As it turns out, it is, and that's what formal verification is all about!

In a nutshell, formal verification is about building *mathematical models* of software, and proving (mathematically) that these models have certain (mathematically defined) properties (correctness). There are many ways to do that (e.g., see [56]). In this course, we will use primarily the TLA+ framework developed by Turing Award winner Leslie Lamport [28, 29]. Not only is TLA+ well-designed for pedagogical purposes, it is also well-suited for concurrent software, the application domain of our interest in this class. TLA+ has also been successfully used in the industry for real-world applications [37]. (In addition to Amazon, TLA+ has been extensively used at Microsoft, where Leslie Lamport worked for many years. I also happen to know that TLA+ is routinely used at MongoDB: my former student William Schultz has worked at MongoDB for many years designing distributed protocols and verifying them with TLA+. Some of these efforts are described in academic papers [46, 44, 47, 45].)

Another well-known framework for modeling and verifying concurrent software is Spin [23, 24]. I haven't decided yet whether we will use Spin in this class, but you are welcome to try it out.

Note that formal verification is not the only formal technique. For example, *static analysis* represents an entire class of program analysis techniques, including some which have been successfully applied to the analysis of smart contracts, e.g. [52, 54]. In this course we will mostly focus on formal verification techniques, although we may also touch upon static analysis. We return to the topic of formal methods in §4.

1.6 Cryptography and Zero-Knowledge

As we shall see in §5.2, cryptography is essential for the correct functioning of the protocols that implement blockchains. Moreover, as we shall see in §**??**, so-called *Layer 2* (L2) blockchains have evolved to address scalability of *Layer 1* (L1) blockchains such as Ethereum. In a nutshell, L2 chains perform some of the computations that would otherwise have to be performed by L1 chains, thereby alleviating the computational burden of the latter. (L1 chains also offload some of their data onto L2s.) This brings however issues of trust: why should L1 trust L2? how can L2 convince L1 that it has performed the computations correctly? and so on. The fascinating science of *zero-knowledge* (ZK) is key in answering some of these questions. We will touch upon cryptography and ZK in §7.

1.7 Beyond DeFi: Digital Democracy?

Blockchains solve fundamental problems of distributed agreements and consensus without the need of trusting a centralized authority. It is therefore natural to wonder whether blockchains can be used in domains where trust is paramount, other that DeFi. (See, for instance, the discussion on *Decentralized Autonomous Organizations* in [11].) Many of our current political processes rely on centralized authorities that are supposed to be trustworthy. But what if we don't trust these authorities? Could we perhaps use blockchains to improve our political processes? We discuss these questions in §8.

1.8 A public visit to `https://etherscan.io/`

One of the great things about blockchains and smart contracts is that a lot of things are publicly available to anyone with internet access. For blockchains, publicity is a must. After all, this is the whole idea of decentralized trust: that the ledger is publicly available for everyone to see. The fortunate thing is that the source code for many smart contracts is also publicly available.[5] This is great news for software researchers and in particular those interested in smart contract analysis, because these researchers are given access to real-world smart contract code.[6]

Let us illustrate this by browsing through a very useful site: `https://etherscan.io/` . Follow the links below and try to understand them (you are encouraged to use the resources listed in §2.4):

- `https://etherscan.io/blocks` : browse through some blocks. You see that some of them are "finalized" whereas others are "unfinalized" (among the latter, some are "safe"). Let's pick a block to zoom in:
- `https://etherscan.io/block/23125063` : click on "88 transactions" to see the block transactions.
- `https://etherscan.io/txs?block=23125063` : click on the first transaction.
- `https://etherscan.io/tx/0x3e052afc368bc02063b1dc087bf0d8564adfba7074dc569e0a9982806b4bc82a` : click on the "To:" address to access the destination contract, and look at its source code.
- `https://etherscan.io/address/0x93ca3db0df3e78e798004bbe14e1ade222b14dfa`

We will be discussing the above (and many more!) in class. By the end of the semester, you should be able to understand what the above mean.

[5] Ethereum does not require this. It only requires the byte code, but not necessarily the source code. Still, the source code for many Ethereum/Solidity smart contracts is publicly available.

[6] Just like sites like github provide a wealth of data that can be used for software engineering and programming language research.

1.9 Industry

You may be wondering whether you could get a job doing things related to the topics of this course. Just I cannot offer financial advice, I cannot offer employment advice either. But I can say that there is a rich ecosystem of institutions and companies active in the field of blockchains and smart contracts, with a focus on security and formal analysis. Ethereum itself is run by the Ethereum Foundation, which among other things provides grants and other kinds of support – `https://esp.ethereum.foundation/`. For example, the 2025 Academic Grants Round invited proposals "across a wide range of disciplines, including Economics & Game Theory, Theoretical and Applied Cryptography, Consensus and Protocol Design, Networking & P2P, Client Engineering, Security, Formal Verification, and the Humanities" – `https://esp.ethereum.foundation/academic-grants` .

There are several companies that perform audits, such as Trail of Bits [40] and OpenZeppelin [48]. Several companies internally use various tools (e.g., decompilers, static analyzers, or formal verifiers) while performing such audits. One of these companies is Dedaub [50], the founder of which is Prof. Smaragdakis with whom I had the pleasure to work recently – c.f. §9. In 2022, Dedaub was awarded a *bug bounty* of $1 million for discovering a major vulnerability in Multichain/AnySwap [51].

Other companies started by academics and specializing in formal analysis techniques include: ♠

- Certora, founded by Prof. Mooly Sagiv [41] – c.f. §**??**.
- Veridise, founded by Prof. Isil Dillig – `https://veridise.com/`.
- Runtime Verification and Pi Squared, two companies founded by Prof. Grigore Rosu – `https://runtimeverification.com/` and `https://pi2.network/`.
- Common Prefix employs several academic researchers and engineers – `https://www.commonprefix.com/`.

2 What we will be doing in this class

This course is given for the first time and there's room to influence its design. What we will be doing in this class will partly depend on your interests. We will discuss in class and I'm open to your suggestions. The plan below is tentative.

My current goals for this course are the following:

- We will learn to read various types of programs and codes and understand their meaning. By the end of the course, you should be able to (non-exhaustive list):
  - Read and understand Solidity programs.
  - Read and understand TLA+ specifications.
  - Browse through websites such as `https://etherscan.io/` and understand what you're seeing there (blocks, transactions, gas, etc).
  - Be able to search effectively for information regarding blockchains and smart contracts.
- We will learn how to write smart contracts in Solidity. By the end of the course, you should also be able to execute and test Solidity programs using Foundry or REMIX.
- We will learn how to write protocol specifications in TLA+ and/or Spin.
- We will learn the basics of formal verification. By the end of the course, you should be able to perform basic verification tasks using TLA+ and/or Spin model-checkers.
- **Research**: there are many "hot" research topics in this area and there are several opportunities to do research. Ideally, you would be writing a paper to be submitted to a conference, or at least good enough for arXiv.org.

I'm expecting a high degree of in-class participation from students. We will be reading papers and other material every week, and also have other assignments. We will be choosing leaders for each paper/assignment. We will all read the paper/do the assignment, but the leaders will lead the discussion. Every one is expected to participate in the discussion.

2.1 Tools you will need

Partial list, under construction:

- Foundry [3].
- REMIX [4].
- TLA+ [28].

The sooner you access/install these tools and start playing with them, the better. You only need one of Foundry or REMIX, you don't need both.

2.2 Readings

Read the following:

1. Weeks 1-2: Bitcoin [36]; Ethereum [11]; Formal methods at Amazon [37].
2. Weeks 2-3: Solidity documentation [5].
3. Weeks 3-?: ???, TLA+ [29].

2.3 Assignments

In all cases below where you are asked to write a contract or modify an existing contract, you are also expected to test your contract in Foundry or REMIX to ensure that it works correctly.

Do the following:

1. Can you find on Etherscan the transactions of the DAO attack and of the two Parity hacks?
2. Modify the `SimpleStorage` smart contract of §3.1.1 to ensure that only the creator (*Deployer*) of the contract can call its `set` function.
3. We learned that transactions are atomic. But what if execution of a transaction never terminates? Can you write non-terminating programs in Solidity?
4. Write two contracts and have them send ETH to each other using the functions described in §3.3.
5. Explain the meaning of each of the following Solidity constructs: `require`, `public`, `address`, `msg.sender`, `mapping(address => uint)`, `view`, `payable`, `receive`, `msg.value`, `send`, `transfer`, `fallback`, `block.timestamp`, `tx.origin`, `address(this).balance`, `msg.data`, `block`, ... TBC

2.4 Resources

This course covers many topics and there is an overwhelming amount of offline and online documentation and other resources relevant to all these topics. I will be listing relevant documentation at the end of each section covering each of these topics. These lists will necessarily be non-exhaustive, and students are invited to contribute! If you found something useful please feel free to suggest it. ♠

Let us list here some websites which we will be using frequently:

- Etherscan [2].
- The Solidity documentation website [5].
- The Ethereum website [1] and especially https://ethereum.org/en/developers/docs/ .
- Lamport's TLA+ website [28].

Many companies maintain extensive websites with documentation and blogs that are often quite useful, e.g.:

- OpenZeppelin documentation site: https://docs.openzeppelin.com/. OpenZeppelin also provides source for many contracts, including standards like ERC20: see https://docs.openzeppelin.com/contracts/.
- Dedaub blog: https://dedaub.com/blog/ . Dedaub also maintains a website offering various tools including a contract library, decompiler, static analysis, etc: https://app.dedaub.com/.
- Certora blog: https://www.certora.com/blog .

2.4.1 On the use of AI in this class

It's OK to use AI tools like ChatGPT in this class, *provided that*. Provided that what? It depends on the use that you make. The ultimate rule is that I will consider you responsible for whatever you provide, and you have to assume that responsibility. For example, questions will arise in class, that we cannot answer on the spot, say, and someone will be assigned to find the answer to such a question and present it in the class next time. That person can use AI to find the answer. But they cannot just present us the answer and say *AI told me that*. Why not? Because AI is not trustworthy. It is not a reliable scientific source. Maybe AI gave the right answer. It is your responsibility to verify this and back-up the answer with reliable references. Reliable sources are primarily scientific papers. These are peer-reviewed by other scientists so they have a high-degree of trustworthiness. I will accept other sources as well, provided we all agree that they are trustworthy. For example, you may cite a youtube video if it's a video of a talk by a scientist (who has a web page or papers online, etc). You may also cite online accepted documentation, for example, https://docs.soliditylang.org/en/latest/index.html. You may also quote Wikipedia pages as long as the passages you quote have themselves trustworthy references.

I advise against using AI to write text for you, e.g., write a report or even a paragraph. The reason is that I consider learning to write as fundamental as learning to think. In fact, I believe that the two are inextricably linked. If you cannot structure your thoughts about whatever topic and write these thoughts down, then you have no coherent thoughts at all. You may have some vague ideas, but they are messy, sloppy, or incoherent, and therefore cannot be considered serious thoughts. Still, I will not forbid the use of AI for writing text, because it can be useful in formulating things in a language you do not master, etc. Ultimately, however, the rule above applies: *you assume responsibility on whatever you write*, independently of whether you used AI or not. You will sign your writings with your name, and I consider them to be yours.

Having said that, signing with your name verbatim AI-generated text is not allowed: I consider that plagiarism (see below). When you copy-paste test from ChatGPT and you use it verbatim, you plagiarize ChatGPT. You might think that that's OK because ChatGPT is not human and so who cares? But the point is that by signing your name on such a text you claim credit for something which is not yours. If you want to use ChatGPT text verbatim, you can quote it. But then the above rules apply: I don't trust what ChatGPT generates, unless I can trace it back to (non-AI) reliable sources. If you can do that, you should instead quote these sources directly.

2.4.2 On plagiarism

Note that assuming responsibility about your writings does not exempt you from adhering to some basic rules of academic conduct. On of those rules is that plagiarism is forbidden. Plagiarism roughly means copying

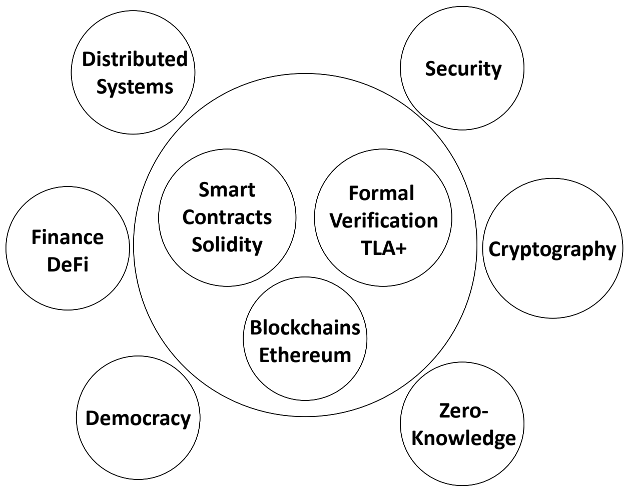

Figure 1: Core and peripheral topics in this course.

others without crediting them (referencing them or citing them properly). Please familiarize yourself with the rules from the relevant university resources on academic conduct.

2.5 Summary

You can look at this course as an introduction to several topics (Figure 1). First, to the world of DeFi, blockchains, and smart contracts. Second, to the world of formal analysis, formal methods and verification. Both these worlds are fascinating enough on their own. Their intersection is even more fascinating. There are other fascinating topics as well, such as zero-knowledge proofs, which we will also touch upon. Then there is the world of finance and money which we will also get a glimpse at, from the DeFi point of view. Any finally there is the world of political systems and democracy, which is also related to blockchains, zero-knowledge, decentralized decision-making and decentralized trust. I hope you will enjoy this course.

3 Smart Contracts

The are several blockchains out there offering ways to program smart contracts. In this course, we focus on Ethereum and Solidity. Ethereum is a blockchain. Solidity is a programming language. You can think of Solidity programs as running *on top of* Ethereum. Specifically, Solidity programs are compiled into bytecode which runs on the *Ethereum Virtual Machine* (EVM). We will explain what all these mean in the sequel. For now, retain that for us a smart contract will be a Solidity program that runs on Ethereum.

Extensive documentation on Solidity is available at [5]. **You should be trying the Solidity programs below (and others that you find online) on your own, using Foundry [3] or REMIX [4] – see§ 3.2.**

3.1 Solidity

Solidity is both like and unlike other programming languages you might be used to. It is like other imperative programming languages: it has assignment statements, if-then-else statements, while loops, function calls, and so on. It is also object-oriented: it has inheritance, access modifiers, function modifiers, etc. But Solidity is also unlike traditional programming languages, in several ways:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

contract SimpleStorage {
    uint storedData;

    function set(uint x) public {
        storedData = x;
    }

    function get() public view returns (uint) {
        return storedData;
    }
}
```

Figure 2: A Solidity contract – taken from [5].

3.1.1 Permanent storage on the blockchain

A contract in Solidity is a collection of code (its functions) and data (its state). This is like in other object-oriented programming languages. But unlike in typical languages where the data of an object disappears once the program terminates, or when the computer is switched-off, say, in Solidity, some data (*storage* data) is stored permanently on the Ethereum blockchain.

For example, look at the `SimpleStorage` contract shown in Figure 2. The line `uint storedData;` declares a *state variable* called `storedData` of type `uint` (unsigned integer of 256 bits). The contents of this variable reside on the contract's storage which itself resides at a specific *address* on the blockchain.[7] The `set` function can change the value of the variable, but the history (i.e., the past values) is maintained.

Not all variables of a contract are stored on the blockchain. For example, local variables of functions persist during execution of the function but disappear when the function terminates. The Solidity documentation [5] has an in-depth discussion of the different types of memory in Solidity.

3.1.2 Reactivity – state machines

Typical programs in traditional programming languages have a `main` function that is the first thing called when the program starts. Solidity contracts don't have a `main` function. Solidity contracts are *reactive systems* [33, 34]. They can be seen as *state machines*. A state machine is something that has a *state* and a set of *transitions* that update the state (see §4.3). The state of a contract corresponds to state variables like `storedData`. The transitions correspond to functions like `set` and `get`. The latter can be seen as a transition that leaves the state unchanged. This is just a special case of a general transition that changes the state.

In a state machine, transitions can also have *inputs* and *outputs*. The inputs are provided by the *environment* (the initiator, or "caller" of the transition) and might influence the state update. These inputs correspond to the input arguments of the contract's functions. For example, function `set` takes input `x` of type `uint` and updates the state to `x`. The outputs of a transition are returned to the environment. They correspond to the return values of the contract's functions. For example, function `get` returns the current value of `storedData`.

The above discussion raises some valid questions: *who is the "environment"? who or what exactly calls the functions of the contract?* and *how is a contract created initially?* Playing with Foundry and REMIX

[7]Throughout this section, we use blockchain terms like *address*, *block*, *account*, *transaction*, etc. We will explain these concepts more in detail when we discuss blockchains – c.f. §5. For now, the reader is referred to the subsection *Introduction to Smart Contracts – Blockchain Basics* of [5].

offers answers to most of these questions: see §3.2. For now, we can say that contracts can be created and called either from within other contracts running on the blockchain (these correspond to *contract accounts*) or *externally* (these correspond to *externally-owned accounts*).

3.1.3 Atomicity – transactions – `require` statements

Function calls in Solidity are *atomic*: either they execute to the end, or, if some runtime error occurs they *revert*, meaning they abort *without changing the state of the contract*. The best way to illustrate this is with an example. At the same time, we will also introduce the `require` statement.

Consider the `SimpleStorage` contract that we saw above. Modify its `set` function as follows:

```
function set(uint x) public {
    require( x < 100 );
    storedData = x;
}
```

The `require` statement takes a condition (in this case, `x < 100`). At runtime, the condition is checked: if the condition is satisfied, execution proceeds normally. If the condition is violated, the call reverts. Play with this modified contract to ensure you understand this. What happens if you put the `require` statement *after* the assignment `storedData = x;` ?

The words *atomicity* and *atomic* come from *atom*, which comes from Greek and means *non-divisible*. So an atomic set of instructions cannot be divided: either all instructions execute, or none. In addition, an atomic set of instructions cannot be interrupted: the EVM cannot pause execution of a Solidity function in order to execute something else, and then resume the execution of that function.[8]

Atomicity is very important, especially in financial and database applications, and allows us to speak of *transactions*. A Solidity function call is a transaction, which either updates the state completely, as the programmer intended, or not at all. Just like in a transaction, say, with a bank's ATM, this avoids inconsistencies due to various types of runtime errors and exceptions. For example, imagine that you are attempting to withdraw money from your bank account using an ATM. You asked for \$100, the program running in the ATM updated your account balance by deducting the \$100 from it, and attempts to give you the cash. However, at that point, the machine breaks down, and no cash is given. The entire operation should abort and your account balance should revert to its original value. This is the concept of an atomic transaction, and this is also how Solidity function calls work. For that reason, the terminology often used is *an account sends a transaction* [5, 19].

3.1.4 Gas, ether, wei

In most programming languages, computation is considered to be "free": it takes time, and memory, and it also consumes electricity. But if we ignore the one-time cost of buying our laptop and our electricity bills, we can say that running, say a C program, is free. In Solidity/Ethereum, execution costs: it "burns" *gas* and gas has a price. (Storage on the blockchain also costs gas.) The price of gas fluctuates according to laws of supply and demand (just like the price of real gasoline for cars). Gas prices can be found at the Etherscan website [2] – see also `https://etherscan.io/gastracker` . For example, when I'm writing this, the average gas price reported on Etherscan is 1.077 gwei (per unit of gas). *gwei* stands for *giga wei*, that is, 10^9 wei. *wei* is a subdivision of *ether*. Ether (ETH) is the native (digital) currency of Ethereum. One wei is the smallest subdivision of ETH (like cents to a dollar). One ETH is equivalent to 10^{18} wei, and hence also equivalent to one billion gwei, i.e., 10^9 gwei. In summary:

$$1 \text{ ether} = 1 \text{ ETH} = 1 \text{ billion gwei} = 10^9 \text{ gwei} = 10^9 \cdot 10^9 \text{ wei} = 10^{18} \text{ wei}.$$
$$1 \text{ wei} = 10^{-18} \text{ ETH}.$$
$$1 \text{ gwei} = 10^9 \text{ wei} = 10^9 \cdot 10^{-18} \text{ ETH} = 10^{-9} \text{ ETH}.$$

[8]This is contrary to what happens, for instance, in many operating systems which routinely switch between executing multiple threads or processes on a single-processor machine.

As I'm writing this, one ETH is worth about 4,700 USD [2]. So, assuming a gas price of 1.077 gwei, a unit of gas costs $1.077 \cdot 10^{-9} \cdot 4700 \approx .000005$ USD, i.e., $5 \cdot 10^{-4}$ cents. Assuming that a transaction burns 10,000 gas, such a transaction would cost, at current ETH and gas prices, $1.077 \cdot 10^{-9} \cdot 4700 \cdot 10000 \approx 0.05$ USD, i.e., 5 cents. This is a small amount, but note that the Ethereum blockchain processes over a million transactions each day [2]. Of course, these transactions are from many different contracts.

Gas measures the cost of computation on the Ethereum blockchain. It's like putting a price on every clock cycle that your computer executes. How does this mechanism work exactly, and why would we want to have it? Quoting from subsection *Introduction to Smart Contracts – The Ethereum Virtual Machine – Gas* of [5]:

> "Upon creation, each transaction is charged with a certain amount of gas that has to be paid for by the originator of the transaction (`tx.origin`). While the EVM executes the transaction, the gas is gradually depleted according to specific rules. If the gas is used up at any point (i.e. it would be negative), an out-of-gas exception is triggered, which ends execution and reverts all modifications made to the state in the current call frame."

Note that if the transaction reverts, the gas is not returned to the transaction originator. The latter will pay the gas whether the transaction completes successfully or not. In both case, the gas is "burnt". This is reasonable, since the EVM performed computation in both cases, and computation costs. Quoting again from the subsection above:

> "This mechanism incentivizes economical use of EVM execution time and also compensates EVM executors (i.e. miners / stakers) for their work. Since each block has a maximum amount of gas, it also limits the amount of work needed to validate a block."

3.1.5 Special types

Solidity has some types not found in traditional programming languages. For instance, the `address` type that represents an Ethereum address, and as such does not allow arithmetic operations. Solidity also has *mapping types*, e.g., `mapping(address => mapping(address => uint256))`. As stated in [5]: "Mappings can be seen as hash tables which are virtually initialized such that every possible key exists from the start and is mapped to a value whose byte-representation is all zeros. However, it is neither possible to obtain a list of all keys of a mapping, nor a list of all values."

`https://docs.soliditylang.org/en/latest/types.html` has a comprehensive discussion of Solidity types.

3.1.6 Events – logs

A Solidity program can define *events* and *emit* them. For example, look at contract `Coin` shown in Figure 3. `Coin` defines an event called `Sent`. This event is emitted whenever a `send` transaction takes place, i.e., whenever the `send` function is called successfully (without reverting). Each time an event is emitted, a corresponding log entry is created and stored on the blockchain. Thus, events are a mechanism to keep logs. Events don't modify the state of the emitting contract (or any other contract) and cannot be cannot be "received" by a contract. Events only create log entries on the blockchain. Such logs are useful to external, off-chain applications (e.g., blockchain explorers like Etherscan). These off-chain apps monitor events and gather information about the internal state of contracts which would otherwise be too hard or impossible to obtain.

3.1.7 Interaction with the Ethereum blockchain

Smart contracts written in Solidity are tighly integrated with the Ethereum blockchain. Ethereum concepts such as addresses, accounts, transactions, and balances, are first-class citizens in the Solidity language. These concepts are somewhat more advanced, hence we discuss them later, in §3.3. For now, make sure you read the Solidity docs as recommended below:

```
contract Coin {
    // The keyword "public" makes variables
    // accessible from other contracts
    address public minter;
    mapping(address => uint) public balances;

    // Events allow clients to react to specific
    // contract changes you declare
    event Sent(address from, address to, uint amount);

    // Constructor code is only run when the contract
    // is created
    constructor() {
        minter = msg.sender;
    }

    // Sends an amount of newly created coins to an address
    // Can only be called by the contract creator
    function mint(address receiver, uint amount) public {
        require(msg.sender == minter);
        balances[receiver] += amount;
    }

    // Errors allow you to provide information about
    // why an operation failed. They are returned
    // to the caller of the function.
    error InsufficientBalance(uint requested, uint available);

    // Sends an amount of existing coins
    // from any caller to an address
    function send(address receiver, uint amount) public {
        require(amount <= balances[msg.sender], InsufficientBalance(amount, balances[msg.sender]));
        balances[msg.sender] -= amount;
        balances[receiver] += amount;
        emit Sent(msg.sender, receiver, amount);
    }
}
```

Figure 3: Solidity contract `Coin` – taken from [5].

3.1.8 Read the Solidity docs

Read the rest of the Solidity docs [5], especially the *Introduction to Smart Contracts* section and the *Blockchain Basics* section. You should be able to understand, for instance, the concepts introduced in the *Subcurrency Example*. Eventually you should also read the *Solidity by Example* section and understand the more advanced examples given there. You don't need to worry about the *Installing the Solidity Compiler* section for now, since we will be using Foundry or REMIX to run our Solidity programs.

At the same time as reading these you should be trying things out with Foundry or REMIX – see §3.2. This holds for these lecture notes as well: you should not necessarily follow the order of the sections, subsections, etc, in the table of contents. A document is necessarily structured linear order. But learning is not linear. You should be jumping back and forth and revisiting things until you internalize them. In addition to jumping back and forth between the Solidity subsection §3.1 and the Foundry/REMIX subsection §3.2, you should be jumping back and forth to other places in these lecture notes, for instance, to §3.3 as well as the Blockchain section §5, which explain concepts such as addresses, accounts, and so on.

3.2 Running Solidity programs

In this course, we take a hands-on, experimental approach to learning. It will therefore be paramount to be able to execute smart contracts written in Solidity. For this, we will be using two platforms: Foundry [3] and REMIX [4]. You don't have to use both. One is enough. But you have to use at least one. Which one is a matter of taste. Foundry is command-line. REMIX runs on your web browser. I use mostly Foundry.

3.2.1 Foundry [3]

I have a Windows laptop, and I'm running Foundry from *Windows Subsystem for Linux* (WSL) – I failed to run it from Cygwin. The instructions below should also be more/less applicable on Linux or Mac computers.

Foundry contains a number of programs (`anvil`, `forge`, `cast`) each doing different things. Typical usage flow:

1. Open two separate Linux/WSL terminals.

2. On one of the terminals, issue `anvil` : this starts a local Ethereum node on your machine. You should see a list of *Available Accounts* along with their corresponding *Private Keys*. Each account is identified by its address, e.g., I see on my machine that account (0) has address `0xf39Fd6e51aad88F6F4ce6aB8827279cffFb92266` and private key `0xac0974bec39a17e36ba4a6b4d238ff944bacb478cbed5efcae784d7bf4f2ff80` . I also see that this account has 10000 ETH to start with. This is nice because we will need (fake) ETH to run and test our Solidity programs.

   You can now pretty much forget about this terminal which is running `anvil` . All the remaining instructions will be launched from the second terminal.

3. On the second terminal we will create and interact with our smart contract. You should first create a Foundry "project" and initialize it. You should probably have some directory called `foundry` or something somewhere in your machine. Go into that directory and issue: `forge init course2025fall` (or choose a name other than `course2025fall` for your forge project). This creates a subdirectory called `course2025fall` (or whatever name you chose for your project) and initializes it with a bunch of files. We won't worry about these files right now.

   You only have to do this `forge init` once per project. You can create separate projects for each smart contract, or just one for this class, or whatever other configuration pleases you.

4. Compile your project: you will do that every time you make changes to a source code file in your project. Go into the directory of your project and issue: `forge build` : you should see compilation happening and a *Compiler run successful!* message in the end. This just compiled the default `Counter.sol` file

that's contained in the initialized project. Populate the `src/` subdirectory of the project with your own `.sol` files (copy them there) and re-compile: run `forge build` again. If the compilation finishes successfully, you're good to go!

For example, you can save the `SimpleStorage` smart contract given earlier into a file called `SimpleStorage.sol` in the `src/` subdir of your project, and re-compile your project.

5. Next, we will deploy our (compiled) smart contract(s) on the blockchain. Issue:
`forge create SimpleStorage  --broadcast --private-key 0x...` On the terminal where you issued the `forge create` command, you should see something like

```
Compiling...
No files changed, compilation skipped
Deployer: 0x70997970C51812dc3A010C7d01b50e0d17dc79C8
Deployed to: 0x8464135c8F25Da09e49BC8782676a84730C318bC
Transaction hash: 0xe1c1bb2b4da71a85839cd648e8e1a183c49e2a875485ca95291d18113c719a59
```

And on the terminal where `anvil` is running, you should also see some interesting stuff happening:

```
...
    Contract created: 0x8464135c8F25Da09e49BC8782676a84730C318bC
    Gas used: 89093
...
    Block Number: 1
    Block Hash: 0x7bc10a2704586dad7a7b381b72367f7a9dae9819f8648e6477ec7d27d2a0a9dd
    Block Time: "Mon, 18 Aug 2025 08:42:08 +0000"
...
```

There's a number of things to note:

- The *Deployer* is the account that *created* the `SimpleStorage` contract. The Deployer account is different from the contract account (see below). The result of `forge create` tells us what the address of the Deployer account is: `0x70997970C51812dc3A010C7d01b50e0d17dc79C8`. This should be one of the addresses listed by `anvil` and should correspond to the private key you used in the `forge create` command. In my case, it is account (1).
- The created contract also has an address: `0x8464135c8F25Da09e49BC8782676a84730C318bC`, reported under *Deployed to* by the `forge create` command, and also under *Contract created* by `anvil`.

6. We can visualize the storage area of the contract we just created (*deployed*) using `cast storage <ADDRESS>` where we pass the address of our contract. Another possible command we could use is `forge inspect`. E.g., I used `forge inspect src/00-SimpleStorage.sol:SimpleStorage storageLayout`.

So far, we have compiled our contract, we have deployed it, and we have looked at the initial value of state variable `storedData` as stored in the contract's storage (that value should be 0). Now we can interact with it, i.e., call its methods.

7. Let's call `get` first. Issue:
`cast call 0x8464135c8F25Da09e49BC8782676a84730C318bC "get()" --private-key 0x...`
where the private key can be any one of the keys of `anvil`. Note that `0x8464135c8F25Da09e49BC8782676a84730C318bC` is the address of our contract. You should see the return value
`0x0000000000000000000000000000000000000000000000000000000000000000`
which is the initial value of `storedData`.

8. Let's now call `set`. Issue:
   `cast send 0x8464135c8F25Da09e49BC8782676a84730C318bC "set(uint)" 1234 --private-key 0x...`
   where again the private key can be any one of the keys of `anvil` (not necessarily the same one you used earlier – why?). Again, note that we are passing the address of our contract. We are also passing `1234` as the argument to the `"set(uint)"` method that we are calling.

   You will notice that we used `cast call` in the case of `get` but `cast send` in the case of `set`. Why? We could have used `cast send` for both: `cast send` creates a *transaction* which is stored in a block of the blockchain. That's why you see things happening in the `anvil` window when you call `cast send`. When you modify the state of a contract, you need to create a transaction. But when you just want to view the state of a contract, you don't have to create a transaction (and by the way, transactions cost gas). Since `get` only reads but does not modify the contract's state variable, we don't need to create a transaction, so we can use `cast call` instead of `cast send`.

9. Now call `get` again: you should see the return value
   `0x00000000000000000000000000000000000000000000000000000000000004d2`
   which is `1234` in hexadecimal. If you don't believe me, you can call `cast` to verify:
   `cast --to-base 0x00000000000000000000000000000000000000000000000000000000000004d2 dec`

10. You can also observe the new value of `storedData` using `cast storage` as above.

3.2.2 REMIX

See https://remix.ethereum.org/ .

3.3 Solidity contract interaction with Ethereum

Solidity programs are tightly integrated with the Ethereum blockchain and can use several of Ethereum's concepts are first-class citizens. We discuss the most important of these in this subsection.

3.3.1 Contract account, address, and balance

When you create (deploy) a Solidity contract, you create an Ethereum *contract account*. In fact, the code of the contract is part of its account.[9] A contract account has an *address*. For example, when you deploy a contract in Foundry using `forge create` you see the address of your newly created contract next to `Deployed to:` – see item 5 of §3.2.1.

Every Ethereum account can hold ether (ETH). The amount of ETH that an account holds at any given point in time is the account's *balance*. You can access the balance of your contract account from within your Solidity program with `address(this).balance`. In general, you can access the balance of an account at address `A` with `A.balance` (`A` must be of type `address`).

From Foundry, you can see the balance of an account with `cast balance`.

3.3.2 Receiving ETH

Ethereum accounts can receive ETH (and hence increase their balance). Contract accounts can receive ETH is several different ways, some of which are (see [5]):

- Via any `payable` functions of the contract, including a `payable constructor` function (if it exists).
- Via the contract's `receive` function (which must always be `payable` if it exists).
- Via the contract's `fallback` function, if the latter exists and is `payable`, and provided the contract does not have a `receive` function. (If the contract has both a `receive` and a `fallback` function, the latter is never called because `receive` is called in its place.)

[9]Ethereum also has another type of accounts, *externally-owned accounts*. Much of our discussion here applies to both types of accounts, but our focus is on contract accounts. See https://ethereum.org/developers/docs/accounts/ for more.

3.3.3 Sending ETH

Ethereum accounts can send ETH (and hence decrease their balance). Contract accounts can send ETH is several different ways, some of which are (see [5]):

- Via `<address payable>.transfer` and `<address payable>.send`.
- Via `<address payable>.call{value: <ether_to_send>}("")`.

3.4 Bugs and attacks

3.4.1 The DAO attack

3.4.2 The Parity Wallet attacks

4 Formal methods and verification

4.1 The science of software

Formal methods is a fascinating topic and the one I spent most of my career on. The term *formal methods* is not great, perhaps, as it emphasizes *what kind* of methods we are talking about, instead of *what these methods are used for*. The term does not tell much to most students. But perhaps the term is revealing as to the mentality of those who practice such methods and who, in my experience, are folks who like math, formalism, and logic. But aside from personal preference and taste, formal methods are fundamental and crucial for software development. In my mind, they are the *science of software*:

- Science is knowledge that can make *predictions*. Predictions can also be thought as *guarantees*. The strongest the science, the strongest the predictions it can make. The science of physics can predict with great accuracy the force required to lift a certain mass to a certain height, and the energy that will be expended. The science of astronomy can predict with great accuracy (guarantee) the next time when Halley's Comet will appear. You can trust this prediction. You can bet money on it. Can you say the same thing for the pseudoscience of astrology, or economics for that matter?
- What predictions can we make about software, that is, about the programs that we write?
- Can we guarantee that our program will not crash? That it won't throw an exception? That it will produce the correct output? And what exactly do we mean by "correct" output? Is the program supposed to work for any input, or only for valid inputs, and what exactly do we mean by "valid" input? Can we guarantee that our program is secure and what does "secure" mean exactly? And so on.
- Formal methods are methods aimed at providing this kind of guarantees. In that sense, these methods constitute the science of software.

You might think, *if formal methods are so important, how come I have never heard of them before?* You might wonder how formal methods are related to the many programming courses you have already taken. Software engineering, programming languages, and the like. These are fundamental, of course. But in the end of the day, the methods covered in such courses do not answer the above questions in a satisfactory way. They cannot make strong enough guarantees. When it comes to program correctness, most software engineering methods boil down to *testing*, that is, running the program enough times on enough inputs. But as famous computer scientist Dijkstra famously quipped, *program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence* [18].

Formal methods is a vast area. It is both vast and deep, which might explain its relative absence from undergraduate computer science curricula. The latter is changing though. More and more universities introduce formal methods into their curriculum. At Northeastern, we have undergraduate course *CS 2800: Logic and Computation* (some other courses related to the topics of CS 2800 are listed in [56]). I am also regularly giving a crosslisted graduate/advanced undergraduate course *CS 7430: Formal Specification, Verification, and Synthesis / CS 4830: System Specification, Verification, and Synthesis*. See my website for the latest edition of these courses. Other profs at Northeastern also give courses either exclusively on formal methods, or with formal methods content in them. There are many textbooks on topics related to formal methods. I will not prescribe a certain textbook here (if you are interested, see [56] for a partial list). We will use the TLA+ book [29] and other material as needed. Remember that this course is not a course about formal methods. Formal methods are a means to an end, not the end. Our goal is the analysis of smart contracts. In terms of what we said above, we want to make predictions/guarantees about smart contracts, and we shall use formal methods for that purpose.

We should point out here that formal methods are not the only ... formal methods out there! They are just one sub-area of the larger area of program analysis, which includes many formal or semi-formal techniques, such as type theory, static analysis, and many more. We will not be covering those. This is not to say they are not relevant to smart contracts. Indeed, there are many program analysis techniques that are routinely applied to smart contracts. There is also active research in areas such as static analysis for smart contracts (e.g., see [22, 53]). This course has been inspired by my contact with Prof. Smaragdakis who does research and teaches courses on static analysis for smart contracts – c.f. §9.

In what follows in this section, we introduce the most fundamental concepts in formal methods.

4.2 Formal specification and verification = formal proofs

Much of formal methods boils down to *proving* that a given program is correct. The keyword here is *proof*, as in mathematical proof. In order to prove anything mathematically, we need to do two things:

- We need to *define* what it is that we are trying to prove. For instance, in math, we need to state a *theorem* or a *lemma* or something like that. This mathematical statement will typically involve some concepts that we need to define formally as well, in order for the statement to make sense. For example, in order for the statement *every polynomial of odd degree has a root* to make sense, we need to define what a *polynomial* is, what its *degree* and *root* are, and so on.
- Once the statement is completely defined, we need to prove it, that is, we need to write a *proof.*

In formal methods we have the same two concepts, under the names *specification* and *verification*:

Specification: This consists in defining formally what we are trying to prove. In our case, it consists in defining formally two things: (1) the program that we are interested in; and (2) what it means for this program to be correct.[10] Typically, the activity of formal specification results in two formal models: (1) a formal description P of the program; and (2) a formal description ϕ of what it means for P to be correct. Both P and ϕ are written in some underlying theory/logic which has itself a formal notion of *satisfaction*, i.e., of what it means for P to *satisfy* ϕ. This is what it means for P to be "correct" (w.r.t. ϕ). In mathematical notation, this is often written as $P \models \phi$.

Verification: This consists in developing a formal proof that the program is correct. In terms of the above, it consists in developing a formal proof of the statement $P \models \phi$.

Refutation and counterexamples: If we manage to prove $P \models \phi$, great. Our job is done and we can take the rest of the day off. But what if we cannot prove $P \models \phi$? Well, it might be because the statement does not hold! That is, it might be that P does *not* satisfy ϕ, written $P \not\models \phi$ (this is the same as $\neg(P \not\models \phi)$, the negation of $P \not\models \phi$, which can be read as *it is not the case that P satisfies ϕ*).

Showing that P does *not* satisfy ϕ (i.e., proving formally $P \not\models \phi$) is interesting in itself, because it tells us either that P is incorrect (i.e., P indeed has bugs), or that ϕ is not what we want. The latter case is

[10]Sometimes people use *formal specification* to mean only part (2), namely, the specification of correctness. Here we follow [29] and use specification to mean the formal definition of both the program and its correctness.

particularly interesting, and as we shall see, arises often. It is often the case that when we try to write down formally what we mean by correctness, we make mistakes! Such mistakes are very educational and are yet another indication of how important it is to have strong, formal guarantees in software.

Moreover, it is sometimes (but not always) easier to *dis*prove $P \models \phi$ (i.e., to prove $P \not\models \phi$) than to prove it. As we shall see in this class, we typically model P as a set of *behaviors* (*executions* or *runs* of a so-called *transition system*, see §4.3 that follows). Then, $P \models \phi$ typically means that all behaviors of P satisfy ϕ (we will see what it means for one behavior to satisfy ϕ). Thus, if we find one behavior of P that does *not* satisfy ϕ, we have proven $P \not\models \phi$. Such a behavior is called a *counterexample*. It's a proof that P does not satisfy ϕ. A counterexample is a very useful thing, because it can help us determine whether P is at fault (and often even help us locate the bug in P) or whether ϕ is at fault (that is, not the right notion of correctness).

It could also be the case that we can neither complete the proof of $P \models \phi$, nor find a counterexample. In this sad situation (which however does arise in practice, due to the difficulty of such proofs) we don't know whether P is correct or not. We might be stuck or at a loss as to how to proceed. We will discuss such situations in this class.

4.3 Formal modeling of programs: transition systems and state machines

Let us say a few things on how to formally model programs, i.e., in terms of the above, how to develop the formal model P. There are many many modeling languages, tools, and formalisms in which P can be written.[11] We will not cover this topic in any depth, as this is beyond the goals of this class. We will use only what we need for our own modeling purposes (e.g., TLA+ and its references [29]). Still, it is important to give here a short description of the foundations.

4.3.1 Transition systems

A *transition system* is something that has *states* and *transitions*. (For that reason, it is sometimes also called *state-transition system*.) Mathematically, we can represent a transition system as a tuple (S, T) where S and T are two sets. S represents the set of states. T represents the set of transitions. But what is a transition? A transition models a "move" or "jump" if you want, from one state to another state. So a transition can be modeled mathematically as a pair of states. For example, if $S = \{s_0, s_1, s_2, s_3\}$, then the pair (s_0, s_1) represents a move from s_0 to s_1. The pair (s_1, s_3) represents a move from s_1 to s_3. The pair (s_1, s_1) represents a move from s_1 back to itself (this is generally allowed). So, in order for T to be a valid set of transitions, T must be a set of pairs of states. That is, T must have the right type. Mathematically, T must be a subset of $S \times S$ (the Cartesian product of S with itself). We write this as $T \subseteq S \times S$.

We typically also need to equip the transition system with a set of *initial states* which captures all the states where the transition system can start at. So, in addition to S and T, we have a set S_0 of initial states. Since initial states are states, S_0 must be a subset of S, that is, $S_0 \subseteq S$.

That's what a transition system is. A triple (S, S_0, T) where S is the set of states, $S_0 \subseteq S$ is the set of initial states, and $T \subseteq S \times S$ is the set of transitions. This type of object looks deceptively simple, but it is very rich. It is also as fundamental to computer science as numbers are to mathematics and physics. We will not discuss further transition systems in this document, and refer to the resources provided in §4.5 for an in-depth discussion.

4.3.2 State machines

A *state machine* can be seen as a *refinement* of a transition system (or equivalently, the transition system can be seen as an *abstraction* of the state machine). It is a refinement in the sense that a transition in a state machine is augmented with extra information, namely, the *input*. In addition, a state machine has *outputs*, either on the transitions, or on the states, dependending on the type of the state machine (Mealy or Moore, see below). Formally, a state machine is a tuple $(S, s_0, I, O, \delta, \lambda)$ where:

[11] For a discussion of the difference between formalism and language, see [10].

- S is the set of states and $s_0 \in S$ is the (unique) initial state.
- I is the set of *inputs*.
- O is the set of *outputs*.
- $\delta : S \times I \to S$ is the *transition function*. It is a function from $S \times I$ to S. That is, δ takes a pair $(s, x) \in S \times I$ and returns $s' = \delta(s, x)$ so that $s' \in S$. In the pair (s, x), $s \in S$ is a state and $x \in I$ is an input. The return value s' is also a state (s' could be the same as s). So what does the transition function tell us? It tells us that if the machine is at state s and it receives input x then it moves to state s'.
- λ is the *output function*. The type (signature) of λ depends on the type of the state machine. There are two types of machines typically encountered in the literature [35, 25, 31]: *Moore* and *Mealy* machines.

  In Moore machines, the output only depends on the state, and therefore λ has type $\lambda : S \to O$. This tells us that while the machine is at state s it outputs $y = \lambda(s)$.

  In Mealy machines, the output depends both on the state and on the input, and therefore λ has type $\lambda : S \times I \to O$. This tells us that while the machine is at state s, its output changes depending on the input. If the input is x_1, then the machine outputs $y_1 = \lambda(s, x_1)$. If the input changes to x_2 (while the state still remains s) then the output changes to $y_2 = \lambda(s, x_2)$. And so on.

We note that our state machines are *deterministic* in the following two senses. First, given the current state and an input, the next state is uniquely defined. Second, the output is also uniquely defined (either by the current state, or by both the current state and the current input). Our state machines are also *complete* in that the functions δ and λ are both *total* (and not *partial*) functions. This means they are always defined (i.e., they are defined for all elements of their respective domains). This in turn means that the next state is always defined, and also that the output is always defined.

State machines were originally used to model things like electronic circuits (microchips) and since such systems are finite-state, they can be modeled by finite-state machines (FSMs) [35, 25, 31]. However, the concept of a state machine is fundamental and very general. State machines can be used to model pretty much every dynamical system (although we may sometimes need to use nondeterministic or probabilistic machines, which we will not worry about in this course). In particular, state machines are very useful to conceptualize several things that are at the heart of our course. For example, an object in an object-oriented programming language can be seen as a state machine (not necessarily finite-state). The states of the machine correspond to all the possible assignments of values to the state variables: the state variables are the private and public variables of the object. The initial state is determined by what values the constructor assigns to those variables. The transitions correspond to the methods of the class, which update the state variables. The outputs can be seen as the public state variables, as well as the return values of the methods.

4.4 Formal verification

4.5 Readings and other resources on formal verification

There are several books and other resources on formal verification. In addition to the material by Lamport on TLA+ [29, 28] and Holzmann's books on Spin [23, 24], there are books focusing on model-checking and explaining concepts such as transition systems and temporal logics, e.g. [33, 34, 8]. Formal verification is closely tied to the study of logic, and encompasses techniques other than model-checking, for instance, theorem-proving. The lecture notes of my CS-2800 course provide many references to those topics [56].

5 Blockchains

A blockchain like Bitcoin or Ethereum is a distributed system, that is, a collection of computers, typically distributed geographically and connected via some network (in the case of Bitcoin and Ethereum the network is the Internet). Each of the computers runs a copy of the blockchain protocol, that is, the software that implements the blockchain. (The *honest* nodes are supposed to all run the correct protocol. The dishonest/malicious or simply faulty nodes might be doing other things. More on this in §5.2.) The blockchain protocol does two things:

- It executes all the functions necessary to maintain the blockchain itself. The blockchain is what solves the distributed consensus problem. This aspect of blockchains is discussed in §5.2.
- It provides to the users of the blockchain all the promised services. This aspect is discussed in §5.1.

We typically use the term *node* to mean the physical computer (the hardware machine) plus the software that runs on that computer. For example, an Ethereum node is a computer that runs the *Ethereum client*. The Ethereum client is a software implementation of the Ethereum protocol.

5.1 Blockchains: the user perspective

5.2 Blockchains: the science under the hood

At the heart of blockchains is distributed agreement: the set of nodes participating in the blockchain must agree which, among the proposed blocks, is going to be the next block in the chain. In computer science literature, distributed agreement is also called distributed *consensus*. The consensus problem is both fundamental and hard. It is fundamental because almost all computing systems are distributed: not just the internet and the cloud, but also computing that could in principle be done on a single machine is often distributed on multiple machines in order to defend against single-machine failures. And distributed systems must often reach consensus: for instance, a distributed database of a bank must have a consistent view of the bank's accounts.[12]

The consensus problem would be easy if the network never failed, no nodes ever failed, and all nodes were "honest" (i.e., secure, non-compromised, not hacked, non-malicious, and so on). But honesty and absence of failures are unrealistic assumptions to make in the real world, which is why the consensus problem is hard. In fact, it is among the hardest problems in computer science. It has occupied some of the greatest computer scientists in the past [30] and it is still an active area of research. Consensus is a fascinating topic which deserves not one, but many courses devoted exclusively to that topic. In this course we will barely scratch the surface. Interested students who wise to study consensus more in-depth are referred to specialized courses and reading material: see §5.3. Our discussion here draws from all these resources.

5.3 Readings and other resources on blockchains

The science of blockchains belongs to the science of distributed protocols. There are many classic books on distributed protocols, for instance [32]. Turing Award winner Leslie Lamport has done a lot of foundational research not just on formal methods but also on distributed systems, and it's worth reading some of his classic papers on the topic, e.g., [27].

Prof. Tim Roughgarden has a lot of material available online in the web pages of his courses *Foundations of Blockchains* – `https://timroughgarden.github.io/fob21/` – and *The Science of Blockchains* – `https://timroughgarden.org/s25/` . Roughgarden also has a list of video lectures available on youtube: `https://youtube.com/playlist?list=PLEGCF-WLh2RLOHv_xUGLqRts_9JxrckiA` .

See also Prof. Elaine Shi's online book *Foundations of Distributed Consensus and Blockchains* [49].

[12] [49] mentions aircraft control as another application, which in fact motivated original research in distributed fault tolerant computing.

6 Decentralized Finance – DeFi

The birth of cryptocurrencies can be considered to have happened in 2008 with the introduction of Bitcoin [36].[13] Many other cryptocurrencies followed. But cryptocurrencies ("money without banks") are not the only applications of blockchains and smart contracts in the domain of finance. There are many more, such as *liquidity pools*, *automated market makers* (AMMs), decentralized exchanges (DEXs), MEV bot networks, ♠ and many others. Together all these applications constitute the world of Decentralized Finance (DeFi). In this section we explore some of the fundamental concepts of DeFi, starting with even the most basic questions.

6.1 What is money?

What is money? I could answer *money is something that has value*. But that's a somewhat tautological definition, because it begs the next question: *what is value?* Perhaps a better definition is this: *money is something that can be exchanged*. The keyword is *exchange*, which implies some sort of *transaction* where you give money and you get something else. So the value of money is really what transactions/exchanges we can do with it.

If you look at what banks say about this question [6, 9, 38], you will see that they list three basic functions of money: (1) money as a medium of exchange, (2) money as a store of value, and (3) money as a unit of account. Medium of exchange is what we said above. Store of value is again tautological: money has value, therefore can be used to store value. Unit of account means that money provides a common base for comparing prices of things. This again follows from the fact that money has value. And value means capacity for exchange.

So in the end, it all boils down to money being something that can be exchanged. But exchanged for what? It could be exchanged for something "real" (a *commodity* such as bread or wood or gold or a bicycle or real estate). It could be exchange also for some other thing that itself has value (i.e., capacity to itself be exchanged), that is, some other kind of money. For example, USD can be exchanged for EUR.

6.2 Money and trust

Some things like food or wood have intrinsic value to humans. You can eat the food and you can use wood for many purposes. *Fiat* money like USD or EUR does not have any intrinsic value. Yet it has value in the sense of capacity of exchange. What gives (fiat) money this value? Why do I believe that I can exchange USD for something else? The answer is: *because everyone else also believes that* (or at least so I think).

Here's how banks put it: "money works because people believe that it will"[6]; "Money [...] has a value that people trust"[9]; "It is up to a government to decide the value of its fiat money and to regulate its supply. This system relies on public trust in the government and its management of the economy, rather than on the set value of a physical asset."[38] [14]

So money is a social convention. Its value comes from a common belief in it (an *agreement* or *consensus* if you prefer). Therefore, money is fundamentally a matter of trust. Not just individual trust (*I trust the government or my bank*) but also *collective* trust (*I trust that everyone else also trusts the government and the banks*). When trust erodes the entire system is in danger of collapsing, as many of us who have lived through the recent financial crises know.

[13]However, there were several ideas and attempts before that: for instance, see `https://en.wikipedia.org/wiki/Cryptocurrency` under *History*, and `https://www.investopedia.com/tech/were-there-cryptocurrencies-bitcoin/`. It is interesting to note that the *wei* denomination of ETH is named after Wei Dai, a computer scientist who proposed *b-money* as "a scheme for a group of untraceable digital pseudonyms to pay each other with money and to enforce contracts amongst themselves without outside help" [13].

[14]The last statement is somewhat misleading: first, it is not the government that regulates the supply of money but ostensibly independent central banks (like the Federal Reserve in the US and the European Central Bank in the EU. But central banks are not the only entities that are allowed to create ("print") money: commercial banks also create money by issuing loans. According to `https://en.wikipedia.org/wiki/Money_creation`, "the majority of the money supply that the public uses for conducting transactions is created by the commercial banking system in the form of commercial bank deposits. Bank loans issued by commercial banks expand the quantity of bank deposits."

6.3 The double-spending problem

Suppose Alice owns a digital coin and wants to transfer it to Bob, in exchange for one of Bob's delicious pies. Bob gives Alice the pie, and Alice transfers the digital coin to Bob's account, somehow. Bob is happy. But since the coin is digital, not physical, nothing prevents Alice to pretend that she still owns it. Indeed, Alice can go to Chris, get one of his delicious cookies, and transfer the digital coin to Chris' account, following the same process as she did with Bob. Who owns Alice's coin now? Bob or Chris? And what prevents Alice from spending her digital coin as many times as she likes?

This *double-spending problem* (see [36], section *Transactions*) does not arise with physical currency. If I have a physical dollar and I give it to you, I don't have it anymore, so I cannot double-spend it. The problem does not arise with digital means of payment that go through a centralized point, either. Consider, for instance, your credit card. Although you might have a physical card in your wallet (and these days you might not even have that, instead having only a digital card on your phone) you don't give your card when you pay for something. You tap your card, and keep it for the next transaction. However, you cannot double-spend the money of your card, because every transaction needs to be approved by your bank (or Visa, or Mastercard, etc). If you have \$100 left in your card's limit and you buy \$100 worth of goods, then quickly try to re-use your card again to buy another \$100 worth of something, you will fail. The second transaction will not be approved, because all transactions are processed by the same centralized authority (your bank, or Visa, etc). By the time the second request arrives, the centralized authority knows that you have reached your card's limit, and denies your request.

But note that we want to have digital money *without* a centralized authority. How can we then solve the double-spending problem? One idea is to *broadcast* (i.e., publicly announce to everyone) all transactions [13, 36]. So, in the example above when Alice pays Bob a message is sent to everyone saying *Alice paid Bob her digital coin.* Chris sees this message, so when Alice goes to him and tries to double-spend her coin, Chris is not fooled: he knows the coin no longer belongs to Alice but to Bob. The problem with this idea is that it assumes instantaneous and reliable broadcast, which does not exist in the real world. Real networks have delays, and may occasionally also lose messages (lost messages can be retransmitted, which adds further delay). In our example, the message *Alice paid Bob her digital coin* might not arrive at Chris until *after* Alice buys something from Chris. Indeed, as Lamport explained in his famous paper [27], an order of events based on time does not apply in a distributed system: the fact that Alice paid Bob *before* she paid Chris is something that Alice knows but neither Chris nor Bob know.

Therefore, we seem to be stuck. The solution, once again, is *consensus.* We need a mechanism (a *protocol*) by which a set of distributed parties can agree on a unique order of transactions. Blockchains are such a mechanism.

The double-spending problem is a wonderful illustration of both (1) the novel challenges that arise from trying to build digital money without a centralized authority (double-spending), and (2) how overcoming these challenges leads to solving fundamental and difficult problems in computer science and distributed systems.

6.4 The future of cryptocurrencies and DeFi?

What is the future of cryptocurrencies and DeFi in general? Nobody knows. It is worth noting that cryptocurrencies are illegal in many countries today: see `https://en.wikipedia.org/wiki/Legality_of_cryptocurrency_by_country_or_territory`. They are legal in the EU and in the US, but are not considered official *legal tender*, meaning that although you can buy and sell Bitcoin, say, you cannot pay your taxes or your fines with Bitcoin. An interesting case is that of El Salvador: the country made Bitcoin a legal tender in 2021 (and seems to be the only country to have done so), but decided it will no longer accept tax payments in Bitcoin in 2025 – see `https://en.wikipedia.org/wiki/Bitcoin_in_El_Salvador`.

Another interesting case is that of Facebook's cryptocurrency *Diem* (originally called *Libra*) which was abandoned in 2022 – see `https://en.wikipedia.org/wiki/Diem_(digital_currency)`. Within the Diem project, the programming language *Move* was created by computer scientist David L. Dill and his team [19]. Notably, although Diem is dead, Move seems to still be alive – see `https://github.com/move-language/` ♠

move.

7 Cryptography and Zero-Knowledge

8 Decentralized/Digital Democracy etc

9 Acknowledgments

The idea for this course came during my sabbatical in 2024-2025 at the University of Athens, Greece. I would like to thank Prof. Yannis Smaragdakis for hosting me there. Prof. Smaragdakis is an expert in static analysis and smart contracts, and this course grew out of my discussions with him and his group. I was also inspired by Prof. Smaragdakis' course *M228 Analysis of Smart Contracts on Blockchain Platforms* which I partly attended in the Spring of 2025: https://yanniss.github.io/M228/. The two courses are different in that M228 focuses on static analysis methods, while our course here focuses on formal verification methods such as model-checking.

References

[1] Ethereum. https://ethereum.org/en/ and https://ethereum.org/en/developers/docs/.

[2] Etherscan. https://etherscan.io/.

[3] Foundry. https://getfoundry.sh/.

[4] Remix. https://remix.ethereum.org.

[5] Solidity. https://docs.soliditylang.org/en/latest/index.html.

[6] Irena Asmundson and Ceyda Oner. What Is Money? Available at https://www.imf.org/external/pubs/ft/fandd/2012/09/basics.htm. International Monetary Fund.

[7] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A Survey of Attacks on Ethereum Smart Contracts SoK. In *Proceedings of the 6th International Conference on Principles of Security and Trust - Volume 10204*, page 164–186, Berlin, Heidelberg, 2017. Springer-Verlag.

[8] C. Baier and J.-P. Katoen. *Principles of Model Checking*. MIT Press, 2008.

[9] European Central Bank. What is money? https://www.ecb.europa.eu/ecb-and-you/explainers/tell-me-more/html/what_is_money.en.html.

[10] David Broman, Edward A. Lee, Stavros Tripakis, and Martin Törngren. Viewpoints, Formalisms, Languages, and Tools for Cyber-Physical Systems. In *6th International Workshop on Multi-Paradigm Modeling (MPM'12)*, 2012.

[11] Vitalik Buterin. Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform, 2014. Available at https://ethereum.org/en/whitepaper/.

[12] Stefanos Chaliasos, Jens Ernstberger, David Theodore, David Wong, Mohammad Jahanara, and Benjamin Livshits. SoK: What don't we know? Understanding Security Vulnerabilities in SNARKs, 2024.

[13] Wei Dai. b-money, a scheme for a group of untraceable digital pseudonyms to pay each other with money and to enforce contracts amongst themselves without outside help. Available at http://www.weidai.com/bmoney.txt.

[14] Dedaub. The Cetus AMM $200M Hack: How a Flawed "Overflow" Check Led to Catastrophic Loss. https://dedaub.com/blog/the-cetus-amm-200m-hack-how-a-flawed-overflow-check-led-to-catastrophic-loss/, 23 May 2025.

[15] Dedaub. Bedrock vulnerability disclosure and actions. https://dedaub.com/blog/bedrock-vulnerability-disclosure-and-actions/, 26 September 2024.

[16] Dedaub. The $11M Cork Protocol Hack: A Critical Lesson in Uniswap V4 Hook Security. https://dedaub.com/blog/the-11m-cork-protocol-hack-a-critical-lesson-in-uniswap-v4-hook-security/, 30 May 2025.

[17] Kevin Delmolino, Mitchell Arnett, Ahmed E. Kosba, Andrew Miller, and Elaine Shi. Step by Step Towards Creating a Safe Smart Contract: Lessons and Insights from a Cryptocurrency Lab. *IACR Cryptol. ePrint Arch.*, page 460, 2015.

[18] Edsger W. Dijkstra. The Humble Programmer. ACM Turing Lecture 1972. Available at https://www.cs.utexas.edu/~EWD/transcriptions/EWD03xx/EWD340.html.

[19] David L. Dill, Wolfgang Grieskamp, Junkil Park, Shaz Qadeer, Meng Xu, and Jingyi Emma Zhong. Fast and reliable formal verification of smart contracts with the move prover. In Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I*, volume 13243 of *Lecture Notes in Computer Science*, pages 183–200. Springer, 2022.

[20] Christof Ferreira Torres, Antonio Ken Iannillo, Arthur Gervais, and Radu State. The eye of horus: Spotting and analyzing attacks on ethereum smart contracts. In *Financial Cryptography and Data Security: 25th International Conference, FC 2021, Virtual Event, March 1–5, 2021, Revised Selected Papers, Part I*, page 33–52, Berlin, Heidelberg, 2021. Springer-Verlag.

[21] Dor Geyer. Securing Protocols During Development - From a High Level Invariant to a Pool-Draining Vulnerability in SushiSwap's Trident. https://medium.com/certora/exploiting-an-invariant-break-how-we-found-a-pool-draining-bug-in-sushiswaps-trident-585bd98a4d4f. Note: the link to the code provided in the article is no longer valid. Use this instead: https://github.com/sushiswap/trident/blob/master/contracts/pool/constant-product/ConstantProductPool.sol.

[22] Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. MadMax: analyzing the out-of-gas world of smart contracts. *Commun. ACM*, 63(10):87–95, September 2020.

[23] G. Holzmann. *Design and Validation of Computer Protocols.* Prentice Hall, 1991.

[24] G. Holzmann. *The Spin Model Checker.* Addison-Wesley, 2003.

[25] Z. Kohavi. *Switching and finite automata theory.* McGraw-Hill, 2 edition, 1978.

[26] John Kolb, John Yang, Randy H. Katz, and David E. Culler. Quartz: A framework for engineering secure smart contracts. Technical Report UCB/EECS-2020-178, UC Berkeley, Aug 2020.

[27] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[28] Leslie Lamport. My TLA+ Home Page. https://lamport.azurewebsites.net/tla/tla.html.

[29] Leslie Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002. Available at https://lamport.azurewebsites.net/tla/book.html.

[30] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, July 1982.

[31] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines - A survey. *Proceedings of the IEEE*, 84:1090–1126, 1996.

[32] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[33] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, New York, 1991.

[34] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, New York, 1995.

[35] E.F. Moore. Gedanken-experiments on sequential machines. In *Automata Studies*, number 34. Princeton University Press, 1956.

[36] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. Available at https://bitcoin.org/bitcoin.pdf, 2008.

[37] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. How Amazon Web Services Uses Formal Methods. *Commun. ACM*, 58(4):66–73, March 2015.

[38] Bank of England. What is money? https://www.bankofengland.co.uk/explainers/what-is-money.

[39] Marco Ortu, Giacomo Ibba, Giuseppe Destefanis, Claudio Conversano, and Roberto Tonelli. Taxonomic insights into ethereum smart contracts by linking application categories to security vulnerabilities. *Scientific Reports*, 14(23433), 2024.

[40] Alexander Remie, Dominik Teiml, and Josselin Feist. Uniswap V3 Core Security Assessment. Trail of Bits audit report, available at https://www.trailofbits.com/documents/UniswapV3Core.pdf, March 12, 2021.

[41] Mooly Sagiv. Five Myths about Formally Verifying Smart Contracts. https://medium.com/certora/five-myths-about-formally-verifying-smart-contracts-e9a85868e89, Dec 21, 2022.

[42] Dimitri Saingre, Thomas Ledoux, and Jean-Marc Menaud. The cost of immortality: A Time To Live for smart contracts. In *IEEE Symposium on Computers and Communications (ISCC)*, pages 1–7, 2021.

[43] Sarwar Sayeed, Hector Marco-Gisbert, and Tom Caira. Smart Contract: Attacks and Protections. *IEEE Access*, 8:24416–24427, 2020.

[44] William Schultz, Ian Dardik, and Stavros Tripakis. Formal verification of a distributed dynamic reconfiguration protocol. In *Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2022, page 143–152, New York, NY, USA, 2022. Association for Computing Machinery.

[45] William Schultz, Ian Dardik, and Stavros Tripakis. Plain and Simple Inductive Invariant Inference for Distributed Protocols in TLA+. In *FMCAD 2022: Formal Methods in Computer-Aided Design*, 2022.

[46] William Schultz, Siyuan Zhou, Ian Dardik, and Stavros Tripakis. Design and Analysis of a Logless Dynamic Reconfiguration Protocol. In Quentin Bramas, Vincent Gramoli, and Alessia Milani, editors, *25th International Conference on Principles of Distributed Systems (OPODIS 2021)*, volume 217 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 26:1–26:16, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[47] William Schultz, Siyuan Zhou, and Stavros Tripakis. Brief Announcement: Design and Verification of a Logless Dynamic Reconfiguration Protocol in MongoDB Replication. In Seth Gilbert, editor, *35th International Symposium on Distributed Computing (DISC 2021)*, volume 209 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 61:1–61:4, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[48] OpenZeppelin Security. Uniswap v4 Core Audit. OpenZeppelin audit report, available at https://blog.openzeppelin.com/uniswap-v4-core-audit, August 27, 2024.

[49] Elaine Shi. *Foundations of Distributed Consensus and Blockchains*. Available at https://www.distributedconsensus.net/.

[50] Yannis Smaragdakis. The CPIMP Attack: an insanely far-reaching vulnerability, successfully mitigated. https://dedaub.com/blog/the-cpimp-attack-an-insanely-far-reaching-vulnerability-successfully-mitigated/, 15 July 2025.

[51] Yannis Smaragdakis. Phantom Functions and the Billion-Dollar No-op. https://medium.com/dedaub/phantom-functions-and-the-billion-dollar-no-op-c56f062ae49f, Jan 24, 2022.

[52] Yannis Smaragdakis, Neville Grech, Sifis Lagouvardos, Konstantinos Triantafyllou, and Ilias Tsatiris. Symbolic value-flow static analysis: deep, precise, complete modeling of ethereum smart contracts. *Proc. ACM Program. Lang.*, 5(OOPSLA):1–30, 2021.

[53] Yannis Smaragdakis, Neville Grech, Sifis Lagouvardos, Konstantinos Triantafyllou, and Ilias Tsatiris. Symbolic value-flow static analysis: deep, precise, complete modeling of Ethereum smart contracts. *Proc. ACM Program. Lang.*, 5(OOPSLA), October 2021.

[54] Yannis Smaragdakis, Neville Grech, Sifis Lagouvardos, Konstantinos Triantafyllou, Ilias Tsatiris, Yannis Bollanos, and Tony Rocco Valentine. Program analysis for high-value smart contract vulnerabilities: Techniques and insights. *CoRR*, abs/2507.20672, 2025.

[55] Justin Thaler. Proofs, Arguments, and Zero-Knowledge. Available at https://people.cs.georgetown.edu/jthaler/ProofsArgsAndZK.html.

[56] Stavros Tripakis. CS 2800 Logic and Computation – Lecture Notes, Fall 2023. Available at https://course.ccs.neu.edu/cs2800f23/lecture-notes.pdf, 2023.

[57] Gavin Wood. Ethereum: A Secure Decentralized Generalized Transaction Ledger. Available at https://ethereum.github.io/yellowpaper/paper.pdf.

[58] Zihan Zheng, Jerry Chen, Ethan Wang, and Jakub Jackowiak. Parity Wallet Hacks: Postmortem. Slides, available at https://tc.gts3.org/cs8803/2023-spring/student_presentations/team7.pdf.