CS 7430/4830
Formal Specification, Verification, and Synthesis
Smart Contract Edition – Spring 2026
Lecture Notes
UNDER CONSTRUCTION

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

2026-03-19, 13:29:46

Abstract

This course covers topics in formal methods: formal specification, formal verification, and formal synthesis. We follow a *formal methods in the field* (FMitF) approach (in the same sense as the NSF program https://www.nsf.gov/funding/opportunities/fmitf-formal-methods-field/) meaning formal methods applied to a certain application domain, or *field*. The field here is smart contracts.

Contents

1 Motivation for smart contracts **4**
1.1 Cryptocurrencies: money without banks 4
1.2 Blockchains: decentralized trust 4
1.3 Smart Contracts: Decentralized Finance (and more) 5
1.4 Cryptography and Zero-Knowledge 6
1.5 Beyond DeFi: Digital Democracy? 6

2 Motivation for applying formal methods to smart contracts **7**
2.1 Security problems and attacks 7
2.2 Mitigations – Audits, Runtime Monitoring, War Rooms, Formal and Other Techniques 7
2.2.1 Formal verification 8
2.2.2 Other formal techniques, e.g., static analysis 8
2.2.3 AI-based techniques 9
2.3 A public visit to https://etherscan.io/ 9
2.4 Industry 9
2.4.1 Bug bounties 10
2.5 Other courses 11

3 This class **11**
3.1 Goals 11
3.2 Resources 12
3.3 Academic integrity 12
3.3.1 On the use of AI in this class 12
3.3.2 On plagiarism 13
3.3.3 On copying code 13

4 Smart Contracts 13
4.1 Solidity 13
4.1.1 Permanent storage on the blockchain 13
4.1.2 Reactivity – state machines 14
4.1.3 Atomicity – transactions – `require` statements 15
4.1.4 Gas, ether, wei 15
4.1.5 Special types 16
4.1.6 Events – logs 16
4.2 Solidity contracts and Ethereum 16
4.2.1 Contract account, address, and balance 18
4.2.2 Receiving ETH 18
4.2.3 Sending ETH 18
4.3 Read the Solidity docs 18
4.4 Running Solidity programs 18
4.4.1 Foundry [4] 19
4.4.2 REMIX 20

5 Bugs and attacks 21
5.1 The DAO attack – Reentrancy 21
5.2 The Parity Wallet attacks – unprotected ownership, constructors and libraries that aren't, lack of initialization, and `delegatecall` 21

6 Formal methods and verification 23
6.1 Software 23
6.2 The science of software 23
6.3 Formal specification and verification = formal proofs 25
6.4 State-transition models: transition systems and state machines 26
6.4.1 State machines 26
6.4.2 Transition systems 28
6.4.3 Symbolic representations of state machines and transition systems: next-state variables 28
6.4.4 TLA+ 30
6.4.5 Syntax vs semantics 30
6.4.6 Nondeterminism 31
6.4.7 PlusCal 32
6.5 Model-checking 32
6.5.1 State assertions 32
6.5.2 Reachable states 34
6.5.3 Explicit-state model-checking – reachability analysis 34
6.5.4 Infinite-state systems and undecidability 34
6.5.5 State explosion 35
6.5.6 Techniques against state explosion 35
6.6 Theorem proving with LEAN 36
6.7 Theorem-proving: lessons learned 37
6.7.1 Expressing correctness precisely is not always trivial 37
6.7.2 Incomplete or inconsistent specifications 38
6.7.3 Formal proofs are non-trivial 38
6.7.4 The how, what, and why of programming 38
6.8 SAT and SMT solvers 38
6.9 Bounded model-checking 39
6.10 Pre-conditions, post-conditions, Hoare triples 39
6.11 Inductive invariants 41
6.12 Safety and Liveness 42

6.12.1 Safety . . . 42
6.12.2 Liveness . . . 43
6.12.3 Fairness . . . 43
6.13 Temporal logic . . . 43
6.14 Logic vs natural language . . . 44
6.15 Synthesis . . . 44
6.16 Other verification methods and topics . . . 44

7 Decentralized Finance – DeFi **45**
7.1 What is money? . . . 45
7.2 Money and trust . . . 46
7.3 The double-spending problem . . . 46
7.4 Ethereum Tokens: money standards . . . 47
7.5 DAOs, Wallets, Flash loans, Leverage, Liquidity pools, AMMs, DEXs, Stablecoins, Bridges, Auctions, Options, ... 48
7.5.1 Leverage . . . 48
7.6 The future of cryptocurrencies and DeFi? . . . 48

8 Blockchains **49**
8.1 The centralized state machine view . . . 49
8.2 The centralized blockchain data structure view: a permanent, public, append-only ledger . . . 50
8.2.1 Why blocks instead of just transactions? . . . 50
8.2.2 Where exactly is the EVM/Ethereum global state stored? . . . 50
8.2.3 Do we actually need the EVM/Ethereum global state? . . . 51
8.2.4 Why is the global EVM/Ethereum state stored as a *modified Merkle Patricia tree*? . . . 51
8.2.5 How are new blocks added to the blockchain? . . . 51
8.2.6 The centralized blockchain view on Etherscan . . . 52
8.3 Decentralization . . . 52
8.3.1 Concurrency and Asynchrony . . . 52
8.3.2 Crash faults, Byzantine faults, and Sibyl attacks . . . 53
8.3.3 Permissioned vs permissionless . . . 53
8.4 Longest-chain Consensus – Proof of work . . . 53
8.4.1 The chain of hashes . . . 53
8.4.2 Proof of work – Miners . . . 54
8.4.3 Allowing forks – the longest chain wins . . . 54
8.4.4 Refining the steps of §8.2.5 . . . 55
8.5 Proof-of-stake . . . 56
8.5.1 Validators and time slots . . . 56
8.5.2 Finality, epochs, and attestations . . . 57
8.6 Theory of distributed systems . . . 57
8.6.1 The Byzantine Generals Problem . . . 58
8.6.2 Impossibility results and tradeoffs: FLP and CAP . . . 58
8.6.3 State Machine Replication in the permissioned setting . . . 59
8.6.4 State Machine Replication in the permissionless setting . . . 59
8.7 Scaling and Layer 2 . . . 60

9 Cryptography and Zero-Knowledge **60**
9.1 Cryptographic hash functions . . . 60
9.2 Digital signatures . . . 61
9.3 Interactive proofs . . . 61
9.4 Arguments . . . 62
9.5 SNARKs and zkSNARKs . . . 62

9.6 Zero-Knowledge 62

10 Acknowledgments **63**

These lecture notes will be evolving as we progress throughout the Spring 2026 semester.

1 Motivation for smart contracts

We begin by introducing and motivating our application domain: smart contracts.

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

[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.

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 §8.

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 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 [141] 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]

[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, if needed. 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.

Ethereum daily transactions number in the millions – https://etherscan.io/chart/tx. There are over a million and a half Ethereum *tokens* [4] with market capitalizations in the billions of dollars – https://etherscan.io/tokens. According to https://coinmarketcap.com/currencies/ethereum/, the market capitalization of Ethereum itself is over 500 billion USD at the time of writing. 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 (e.g., Solana, Avalanche, and Sui, are others).

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 ♠ 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 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. §1.5 and §**??**.

1.4 Cryptography and Zero-Knowledge

As we shall see in §8, cryptographic primitives such as hash functions and signatures are essential for the security of blockchains. Moreover, as we shall see in §8.7, 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 raises 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) offers answers to such questions. We touch upon cryptography and ZK in §9.

1.5 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 [38].) 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 §**??**.

[4] A *token* here refers to an ERC-20 token. We discuss tokens in §7.4. 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.

2 Motivation for applying formal methods to smart contracts

2.1 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 §5.1.
- 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) [142, 67, 173].
- 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 [142, 67, 173]. We discuss both Parity Wallet attacks in detail in §5.2.

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 GlassWorm attacks of late 2025, e.g., see `https://thehackernews.com/2025/12/glassworm-returns-with-24-m` `html`.
- The CPIMP attack which happened in July 2025 [154].
- The $11M Cork Protocol Hack which happened on the 28th of May 2025 [57].
- The Cetus AMM $200M Hack which happened on May 22, 2025 [55].
- The Bedrock vulnerability which was discovered on 26 September 2024 [56].

2.2 Mitigations – Audits, Runtime Monitoring, War Rooms, Formal and Other Techniques

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? There is not a simple, single answer to this question, because there are many things that need to be done. One essential component is developing good coding practices, e.g., by using well-tested patterns – c.f. `https://ethereum.org/developers/docs/smart-contracts/security/`. In addition:

- 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 even formal 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. [66, 135, 150].

The need to examine either the code or the behavior of the smart contract can arise even after the contract is deployed. A number of companies offer products that *monitor* various contract activities and issue alerts when something looks suspicious (e.g., see `https://dedaub.com/product/security-suite/`). When formal correctness properties are available, such runtime monitors can be generated automatically from the correctness specification, a method called *runtime verification* [81]. What happens if an ongoing attack is detected? or when a critical vulnerability is discovered that can potentially be exploited by attackers? Security experts assemble so-called *war rooms*, where they try to find solutions and take actions to stop the attack while it's happening, to alert stakeholders, or to fix the problem before the attackers discover it; e.g., see [154, 56].

This course focuses on *formal* 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 §6.2). There are many formal techniques out there, enough to fill many university courses (see §6.2 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.

2.2.1 Formal verification

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 [163]). In this course, we will use primarily the TLA+ framework developed by Turing Award winner Leslie Lamport [97, 101]. 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 [116]. In addition to Amazon, TLA+ has been extensively used at Microsoft, where Leslie Lamport worked for many years.[5]

Another well-known framework for modeling and verifying concurrent software is Spin [84, 86]. I haven't decided yet whether we will use Spin in this class, but you are welcome to try it out.

2.2.2 Other formal techniques, e.g., static analysis

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 applied to the analysis of smart contracts, e.g. [156, 157]. In this course we will mostly focus on formal verification techniques (§6).

[5] I also happen to know that TLA+ is routinely used at MongoDB: my former student William Schultz has worked at MongoDB for a number of years designing distributed protocols and verifying them with TLA+. Some of these efforts are described in papers [148, 146, 149, 147].

Static analysis techniques are typically approximate, yielding both *false positive* as well as *false negative* results. A false positive rate of 40% means that out of 100 potential vulnerabilities flagged by the static analysis tool, only 60 will turn out to be true vulnerabilities (i.e., true positives). Interestingly, [157] argues that in the domain of smart contracts high false-positive rates may be acceptable as long as the tool can pinpoint high-value vulnerabilities: “In contrast to academic literature in program analysis, which routinely expects false-positive rates below 50% for publishable results, we posit that a useful analysis for high-value real-world vulnerabilities will likely flag very few programs (under 1%) and will do so with a high false-positive rate (e.g., 95%, meaning that only one-of-twenty human inspections will yield an exploitable vulnerability)” [157]. On the other hand, [172] reports that the majority of exploitable bugs “are beyond existing tools”.

2.2.3 AI-based techniques

AI-based techniques are increasingly used in program analysis and verification tasks in various domains, including in the domain of smart contracts, e.g., see [159, 74]. ♠

2.3 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.[6] 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.[7]

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 §3.2):

- `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.

2.4 Industry

You may be wondering whether you could get a job doing things related to the topics of this course. Just as 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,

[6] 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.

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

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 [66, 135], OpenZeppelin [150], Certora, and Dedaub (see also `https://ethereum.org/developers/docs/smart-contracts/security/`). Several of these companies use various internal or external tools while performing such audits. For instance, Trail of Bits uses a Solidity static analyzer called Slither [66]. Dedaub also use a static analyzer (their own) as well as a decompiler (e.g., see [154]).

Dedaub is founded by static analysis expert Prof. Smaragdakis with whom I had the pleasure to work recently – c.f. §10. In 2022, Dedaub was awarded a *bug bounty* of $1 million for discovering a major vulnerability in Multichain/AnySwap [155] (see also §2.4.1). Other companies started by academics and specializing in formal analysis for blockchains include (partial list): ♠

- Certora, founded by formal verification expert Prof. Mooly Sagiv [140].
- Veridise, founded by formal verification expert Prof. Isil Dillig – `https://veridise.com/`.
- Runtime Verification and Pi Squared, two companies founded by formal verification expert Prof. Grigore Rosu – `https://runtimeverification.com/` and `https://pi2.network/`.

But there are many more companies active in the more broad domain of crypto/blockchains, including (very partial list): ♠

- Common Prefix employs several academic researchers and engineers – `https://www.commonprefix.com/`.
- a16zcrypto is a mix investment/engineering/research/policy firm which employs, among others, distributed systems expert Prof. Tim Roughgarden and zero-knowledge expert Prof. Justin Thaler – `https://a16zcrypto.com/`.
- `https://babylonlabs.io/` co-founded by Prof. David Tse [160].
- `https://starkware.co/` co-founded by Prof. Eli Ben-Sasson.
- `https://consensys.io/` and `https://diligence.consensys.io/`
- `https://www.paradigm.xyz/`
- `https://risczero.com/`
- `https://zerosync.org/`
- `https://www.nethermind.io/`

2.4.1 Bug bounties

See, for example: ♠

- `https://immunefi.com/bug-bounty/` and `https://immunefi.com/hackers/`
- `https://www.theblock.co/post/301025/web3-immunefi-ethical-hacker-payouts`

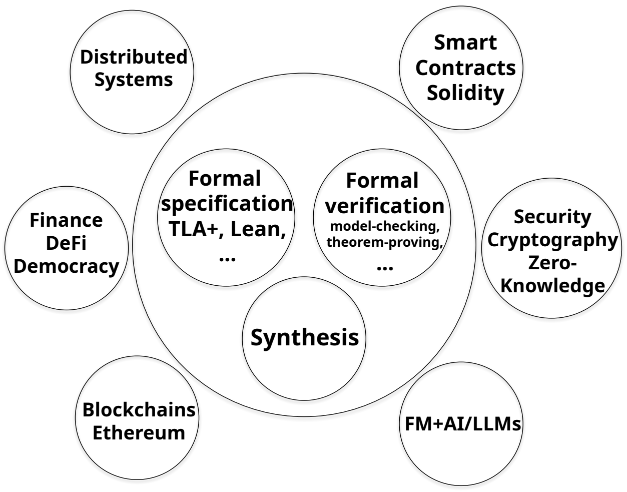
Distributed Systems
Smart Contracts Solidity
Formal specification TLA+, Lean, ...
Formal verification model-checking, theorem-proving, ...
Finance DeFi Democracy
Security Cryptography Zero-Knowledge
Synthesis
Blockchains Ethereum
FM+AI/LLMs

Figure 1: Core and peripheral topics in this course.

2.5 Other courses

I am listing here some other courses which are (somewhat) related to any of the topics of this class and have sufficient material publicly available online. This list is terribly incomplete, and I would be thankful for any omissions you point me to: ♠

- *Software Foundations* by Benjamin C. Pierce et al [9].
- *Blockchain Foundations* by Dr. Dionysis Zindros et al: https://www.marabu.dev/.

3 This class

This class evolved from two classes: (1) the same class which I have been teaching up to Fall 2023 (see https://www.ccs.neu.edu/~stavros/ under Teaching) and (2) the *Special Topics* class CS 4973/6983 *Smart Contracts and Analysis* (SCA) which I taught in Fall 2025: https://www.ccs.neu.edu/~stavros/sca25.html. A previous version of these lecture notes is available at [164]. Compared to the SCA class, this class focuses more on formal methods and less on smart contracts. An overview of the topics we will discuss this semester is given in Figure 1. Some of these topics we will cover in depth, whereas others we will merely touch upon, depending on time and priorities. I hope you will enjoy this course.

3.1 Goals

- Read and understand Solidity programs. Learn how to write smart contracts in Solidity and test them using Foundry or REMIX.
- Read and understand TLA+, Lean, Z3, ..., specifications. Learn how to write such specifications. Learn how to use the corresponding tools (model-checkers, theorem provers, SAT/SMT solvers, ...).
- 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.

- **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.

3.2 Resources

This course covers many topics for which there is an overwhelming amount of offline and online documentation and other relevant resources. 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 find something useful please feel free to suggest it. ♠

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

- Etherscan [3].
- The Solidity documentation website [10].
- The Ethereum website [1] and especially `https://ethereum.org/en/developers/docs/` .
- Lamport’s TLA+ website [97] and PlusCal tutorial [98]. **I recommend downloading `tla2tools.jar` from `https://github.com/tlaplus/tlaplus/releases` and running the TLA+ tools from the command line:** `https://learntla.com/topics/cli.html`. My former student Will Schultz has been developing an interactive simulator for TLA+: `https://will62794.github.io/spectacle/`.
- Lean: [6]
- Z3: [11]

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` .

3.3 Academic integrity

Students are expected to comply with the university’s academic integrity and similar policies: see `https://osccr.sites.northeastern.edu/academic-integrity-policy/`, `https://osccr.sites.northeastern.edu/code-of-student-conduct/`, and so on, from `https://osccr.sites.northeastern.edu/`.

3.3.1 On the use of AI in this class

In accordance with university policy (see `https://policies.northeastern.edu/policy125/` and follow the *Standards for the Use of Generative AI in Teaching* link) AI use in this class is *Sometimes Allowed*. My philosophy is that you learn something only by trying it out yourself, so you should not delegate the “trying out” part to AI. Writing code, papers, slides, etc, are all examples of “trying out”. They are all creative processes that result in learning and should not be delegated to AI. In all cases, I will consider you responsible as the sole author for whatever you provide, and you have to assume that responsibility. Note that putting your name on something created by someone else is plagiarism (see below). I consider putting your name on something created by AI plagiarism, too.

You may consult AI just like you consult, say, google search. But you cannot cite AI. You cannot say "AI told me that". You can use AI to find references (papers, webpages, etc) and then cite those references. (And your references must always be trustworthy.)

3.3.2 On plagiarism

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

3.3.3 On copying code

We all copy code that we find online. The question is how much code and for what purpose. If you are trying to use a new function, e.g., like the `receive` or `call` functions in §4.2, and you don't remember exactly the syntax or signature of the function, you might want to look up an example online. This is OK. **But it is not OK is to copy entire pieces of code**, for instance, entire contracts.

It's not just that such copying is plagiarism. It's also that by doing so you don't learn. You may *think* that you understand the code that you have copied, so what's the big deal if you didn't write it yourself? Since you understand it, you learned, right? This is an illusion. My experience (and that of many others) is that you don't really understand something until you try to do it yourself. You cannot learn to ride a bike by watching someone else ride, and you cannot learn to drive by watching someone else drive. Similarly, you cannot learn to code simply by reading code. You have to also write it, and execute it, and test it. (And ideally also specify it and verify it, but that's another story – the story of §6.)

4 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.[8] 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 [10]. **You should be trying the Solidity programs below (and others that you find online) on your own, using Foundry [4] or REMIX [7] – see§ 4.4.**

4.1 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:

4.1.1 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.

[8] "The Solidity programming language is an open-source, community project governed by a core team. The core team is sponsored by the Ethereum Foundation" – `https://soliditylang.org/about/`.

```
// 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 [10].

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.[9] The `set` function can change the value of the variable, but the history (i.e., the past values) is maintained. Where exactly is this history stored? We discuss this in §8 and in particular §8.2.2. Even though the values of contract state variables are not stored in the blocks of the blockchain, we can still use the `--block` option of `cast call` to retrieve historical values, e.g., as in:

```
cast call ... "get()" --private-key ... --block 1
```

Not all variables of a contract are stored permanently. For example, local variables of functions persist during execution of the function but disappear when the function terminates. The Solidity documentation [10] has an in-depth discussion of the different types of memory in Solidity.

4.1.2 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* [109, 110]. They can be seen as *state machines*. A state machine is something that has a *state* and a *transition function* that updates the state – see §6.4.1. The state of a contract is the current values of its state variables, like `storedData` in Figure 2. The transition function of a contract is defined by the contract's functions, like `set` and `get`. `set` changes the state whereas `get` leaves the state unchanged (this is just a special case of a transition function).

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 or REMIX

[9] 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. §8. For now, the reader is referred to the subsection *Introduction to Smart Contracts – Blockchain Basics* of [10].

offers answers to most of these questions: see §4.4. 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* – EOAs).

4.1.3 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.[10]

Atomicity is very important, especially in financial and database applications, and allows us to speak of *transactions*. A Solidity function call 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.

4.1.4 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 [3] – 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:

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

[10] 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 4,500 USD (on 16 Sept 2025, according to [3]). Assuming that a transaction consumes 21,000 gas, and assuming a gas price of 1 gwei, such a transaction would cost $21000 \cdot 1 \cdot 10^{-9} \cdot 4500 = .0945$ USD, i.e., about 10 cents. This is a small amount, but note that: (1) both gas and ETH prices vary over time, (2) gas consumption per transaction also varies, and (3) Ethereum processes over a million transactions each day [3] (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 [10]:

> "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."

4.1.5 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 [10]: "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.

4.1.6 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 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.

4.2 Solidity contracts and Ethereum

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. We discuss the most important of these concepts next:

```
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 [10].

4.2.1 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.[11] Each 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 (`Deployed to:`) and you also see the address of the `Deployer` – c.f. item 5 of §4.4.1.

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`. Observe how the balance of a sender decreases with every transaction sent: why?

4.2.2 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 [10]):

- 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.)

4.2.3 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 [10]):

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

4.3 Read the Solidity docs

Read the rest of the Solidity docs [10], 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 §4.4.

4.4 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 [4] and REMIX [7]. 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.

[11]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.

4.4.1 Foundry [4]

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. Create a directory called `foundry` or something somewhere in your machine. Go into that directory and issue: `forge init course2025fall` (or choose a name other than `course2026spring` for your forge project). This creates a subdirectory called `course2026spring` (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.

4.4.2 REMIX

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

5 Bugs and attacks

Like any software, smart contracts have bugs. Bugs make smart contracts vulnerable to hacks and attacks. These attacks can come from human actors acting alone or in combination with *malicious contracts* that is, contracts deployed explicitly with the intent of aiding hackers to steal funds.[12] There is a long list of attacks, some of which are very intricate (e.g., see §2.1). There are several sources you can consult to learn ♠ more about the different kinds of attacks.

- Repositories such as
  - `https://github.com/sigp/solidity-security-blog`
  - `https://github.com/nevillegrech/gigahorse-benchmarks/tree/master/vulnerable-sources`
  - `https://durieux.me/projects/smartbugs.html`
- Survey papers, e.g. [21, 59, 91, 142, 67, 88, 119, 43].
- Online articles and blogs such as the ones cited in §2.1.

5.1 The DAO attack – Reentrancy

You deposit your money in a checking or savings account in your bank. Then your bank invests your money, e.g., by giving out loans. But you have no say in these investments. The DAO was a smart contract which acted in essence as a bank (or venture capital fund) where the depositors/investors had decision-making power over the investments that were made, by voting in favor or against such investments. So The DAO combined principles of finance and investment with those of *Decentralized Autonomous Organizations*. A simplified but nice illustration of The DAO's operation is given in this video: `https://vimeo.com/768513838`.

The same video also illustrates the principles behind The DAO attack which occurred on June 17, 2016. Not only did this attack result in a large number of funds stolen, and the demise of The DAO itself; it also ♠ resulted in a controversial *fork* of the Ethereum blockchain. You can read about the history of The DAO on Wikipedia: `https://en.wikipedia.org/wiki/The_DAO`.

The DAO attack is a *reentrancy attack*. Reentrancy occurs when some function `F` can be called again before the current call to `F` finishes. For example, imagine that `F` is called, then in the middle of its execution `F` calls another function `G`, and eventually `G` ends up calling (a new instance of) `F` again. So at this point, the call stack contains the original call to `F`, then `G`, and then the new call to `F`.

There are many online posts that explain why this can be problematic, especially in the context of The ♠ DAO attack. This post is pedagogical `https://blog.chain.link/reentrancy-attacks-and-the-dao-hack/`, albeit "highly simplified" as the author also states. A more detailed but also more complicated description can be found here: `https://web.archive.org/web/20180128074919/http://vessenes.com/deconstructing-thedao-attack` This refers to the actual functions in The DAO's original Solidity implementation, which is available on Github: `https://github.com/TheDAO/DAO-1.0`. (Functions `splitDAO` and `withdrawRewardFor` are implemented in file `DAO.sol`. Function `payOut` is implemented in file `ManagedAccount.sol`.)

In one of your assignments, you will have the opportunity to reconstruct a reentrancy attack similar to the one that took place in The DAO.

5.2 The Parity Wallet attacks – unprotected ownership, constructors and libraries that aren't, lack of initialization, and `delegatecall`

Ethereum *wallets* are apps that manage how users interact with their Ethereum accounts – `https://ethereum.org/wallets/`. The Parity Multisig Wallet was such a wallet which was attacked twice within

[12]One may wonder why would a hacker implement their attack in a contract deployed on Ethereum, instead of launching the attack completely "externally", that is, by calling functions from externally owned accounts? One reason is because some attacks exploit atomicity of transactions, which allows them not to be interrupted during an attack. For example, some attacks utilize *flash loans* which consist in borrowing funds and paying them back in the same transaction §7.5.

a few months, first in July and then again in November 2017. The company Parity still exists (https://www.parity.io/) but unfortunately they seem to have erased all their online announcements and post-mortems of these attacks. Fortunately, there are other online accounts and explanations of these attacks still available, e.g., see [122, 132, 126, 49, 173]. And the history of the attack transactions is available on Etherscan.

The first attack [122, 132] consisted in two steps: (1) attacker takes ownership of the wallet, (2) attacker transfers wallet funds to attacker's account. The attacked source code is available at https://github.com/openethereum/parity-ethereum/blob/4d08e7b0aec46443bf26547b17d10cb302672835/js/src/contracts/snippets/enhanced-wallet.sol and a key problematic function is:

```
  // constructor - just pass on the owner array to the multiowned and ...
  function initWallet(address[] _owners, uint _required, uint _daylimit) { ... }
```

Do you notice something "fishy" about `initWallet`? The comment claims it to be a constructor, but is it really?[13] Also, `initWallet` doesn't have a *function visibility* specifier: is it `public` or `private`? (The *Function Visibility* section of [10] does not mention the default function visibility but online sources claim it to be ♠ public by default.) As it turned out, the attacker gained ownership of the wallet by calling `initWallet` (via `delegatecall`, which we won't worry about right now). We can indeed see the relevant transaction: https://etherscan.io/tx/0x9dbf0326a03a2a3719c27be4fa69aacc9857fd231a8d9dcaede4bb083def75ec. After that, the attacker called the `execute` function to steal the wallet's funds. I don't really understand the logic of ♠ `execute`, but I understand the relevant transaction on Etherscan which shows *Transfer 82,189 ETH*: https://etherscan.io/tx/0xeef10fc5170f669b86c4cd0444882a96087221325f8bf2f55d6188633aa7be7c. Understanding exactly how `execute` works may not be necessary in order to understand one of the keys to this attack: owners of a contract are expected to have great power, hence we can expect that once the attacker gained ownership, the damage was already done. Indeed, the modifier `onlyowner` of `execute` (defined earlier in the contract) didn't help protect the funds, since the attacker was already the owner.

As if the 1st attack was not bad enough, the Parity wallet was attacked again only a few months later [126, 49]. What is remarkable is that the "fixed" Parity wallet contract suffered from the same problem as the original contract: it again allowed an attacker to gain ownership! Despite its bad English and inappropriate language, [126] gives a short and to-the-point account of the 2nd attack, and includes a discussion regarding *who is to blame*. The source code of the "fixed" Parity Wallet which suffered the 2nd attack is available on Etherscan [14] at https://etherscan.io/address/0x863df6bfa4469f3ead0be8f9f2aae51c91a907b4 (click on *Contract*). It contains the following buggy code:

```
// throw unless the contract is not yet initialized.
modifier only_uninitialized { if (m_numOwners > 0) throw; _; }

// constructor - just pass on the owner array to the multiowned and
// the limit to daylimit
function initWallet(address[] _owners, uint _required, uint _daylimit) only_uninitialized { ... }
```

Supposedly, `initWallet` has been "fixed" by adding to it the modifier `only_uninitialized`. The latter is intended to ensure that `initWallet` is only called once, upon initialization, and cannot be called again after that (note that `throw` is now obsolete and has been replaced by `revert`). But upon creation of the `WalletLibrary` contract, its `m_numOwners` is 0. Also, `initWallet` is still publicly callable.[15] Therefore, if

[13]In older versions of Solidity, the constructor was a function that had the same name as the contract, instead of the function defined with the `constructor` keyword used today. But notice that `initWallet` does not have the same name as the contract it belongs to, which is called `WalletLibrary`.

[14]You will note that this contract, as well as many others on Etherscan, is labeled as *Contract Source Code Verified*. What this means is that the source code has been verified to match the bytecode, so that what you can be sure that what you read is really what's being executed (remember, the EVM executes bytecode, not source code). This kind of verification is important, but it has nothing to do with the formal verification we discuss in §6.

[15]Note that the `WalletLibrary` contract is separate from the `Wallet` contracts, and the `m_numOwners` variable of the former is separate from the variable with the same name of the latter.

the attacker is the first to call `initWallet`, they can gain ownership. And that's exactly what the attacker did: https://etherscan.io/tx/0x05f71e1b2cb4f03e547739db15d080fd30c989eda04d37ce6264c5686e0722c9; prior to proceeding in "killing" the `WalletLibrary` contract by calling its self-destruct `suicide` function: https://etherscan.io/tx/0x47f7cff7a5e671884629c93b368cb18f58a993f4b19c2a53a8662e3f1482f690. After being "killed", `WalletLibrary` became inaccessible to all contracts (wallets) that depended on it, thereby effectively making all funds in these other contracts inaccessible (*frozen*). Hundreds of contracts with hundreds of thousands ETH in total were affected [49].

The two Parity attacks illustrate, if nothing else, numerous bad programming practices and oversights. It is also remarkable that the "fixed" `WalletLibrary` was left un-initialized for many months after the first attack.

6 Formal methods and verification

6.1 Software

Our modern societies heavily depend on software, and this dependence is likely to grow. Software is important, and it is also beautifully complex. It is complex first because of its sheer size: estimates in 2015 placed Google's software at about 2 billion lines of code, and Microsoft's Windows operating system at about 50 ♠ million lines of code[16]; in 2017 a pacemaker had about 100 thousand lines of code, the Boeing 787 airplane had more than 10 million, and a high-end car had about 100 million[17]; some estimates place the size of new software produced every year to the hundreds of billions of lines of code[18].

But even very small programs can be extremely complex. The famous *Collatz conjecture* states that the following program terminates for all possible inputs:

```
n := input a natural number;
while (n > 1)
  if (n is even)
    n := n/2;
  else
    n := 3*n + 1;
```

The Collatz conjecture is an open problem.[19] It is a *conjecture* (i.e., something we believe is true), but not a *theorem* (i.e., not something we have proven). The fact that this 6-line program defies the understanding of even our best mathematicians tells us that there is something inherently complex and challenging about software. Software is the most complex artifact that humans have ever constructed. Understanding software is an important intellectual challenge for humanity.

6.2 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

[16]According to https://www.wired.com/2015/09/google-2-billion-lines-codeand-one-place/.
[17]According to https://www.visualcapitalist.com/millions-lines-of-code/.
[18]According to https://cybersecurityventures.com/application-security-report-2017/.
[19]If you solve it, you will become famous. See also: https://www.quantamagazine.org/can-computers-solve-the-collatz-conjecture-20200826/ (thanks to Samuel Lowe for suggesting this article).

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 also trust (and reproduce) chemical experiments. Chemistry is a strong science. Mathematics is the strongest science, in the sense that you can trust its predictions "100%" (what are math's predictions?) Can you trust the pseudoscience of astrology?

- 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, and can be seen as complementary to formal methods. One can view formal methods as "raising the bar", so to speak, in terms of guarantees about program correctness. Most software engineering methods boil down to *testing*, that is, running the program enough times on enough inputs. But as renowned computer scientist Dijkstra famously quipped in his ACM Turing Award Lecture in 1972, "program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence" [60]. But if finding bugs by testing is (relatively) easy, proving their absence is not so easy. In this course, you will get a chance to experience what doing formal proofs is like.

Formal methods is a vast area. It is both vast and deep, which might explain its relative absence from undergraduate computer science curricula.[20] 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 [163]). 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 [163] for a partial list). We will use the TLA+ book [101] 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 [77, 156] and §10).

What about adoption of formal methods in practice/industry? We already briefly talked about this in §2.2.1. In my 30 or so years of experience, industrial adoption of formal methods has been steadily growing. In the past, formal methods were considered intractable, too expensive, and in a nutshell impractical. Influential researchers in the software engineering community damned them [52] (interestingly,

[20] Education is identified (correctly in my opinion) as the major obstacle in the even broader adoption of formal methods in [124].

[52] damns formal methods not just for software but also for mathematics; today, adoption of formal methods is increasing not just by the software but also by the mathematics community, e.g., see §6.6). But if in the past formal methods were used only in highly regulated safety critical applications, e.g., aerospace [138, 139], today they are increasingly used for all kinds of software, e.g., see [116, 171, 73]. An extensive *List of companies that use formal verification methods in software engineering* is available at `https://github.com/ligurio/practical-fm`. In our setting, `https://ethereum.org/developers/docs/smart-contracts/formal-verification/` states: "Formal verification is one of the recommended techniques for improving smart contract security."

6.3 Formal specification and verification = formal proofs

Much of formal methods boils down to *proving* that a given program (sometimes called the **implementation**) 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. 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. Although P and ϕ may be written in different languages, there should be some underlying 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, P *satisfies* ϕ is written $P \models \phi$.

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 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 often model P as a set of *behaviors* (*executions* or *runs* of a so-called *transition system*, c.f. §6.4.2). 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.

6.4 State-transition models: transition systems and state machines

When you verify software using a proof assistant like Lean, your program P looks very much like a mathematical function and the correctness specification ϕ is a theorem you state about this function. This is an elegant way of doing things, as P, ϕ, and even the proof itself are all unified under Lean's powerful logic. In other settings, it is more natural to express P and ϕ differently. For concurrent programs like smart contracts, P is typically modeled as some kind of *state-transition* model. A state-transition model is something that has *states* and *transitions*. Transitions are "jumps" ("moves") between states. There are many variants of state-transition models. The most important classes are state machines (typically deterministic) and transition systems (typically non-deterministic). We explain these concepts next.

6.4.1 State machines

A *state machine* is a typically *deterministic* (see below) state-transition model that has states, transitions, inputs, and outputs. Formally, a state machine is a tuple $(S, s_0, I, O, \delta, \lambda)$ where:

- S is a set of *states* and $s_0 \in S$ is the (unique) *initial state*.
- I is a set of *input symbols* (i.e., possible input values).
- O is a set of *output symbols* (i.e., possible output values).
- $\delta : S \times I \to S$ 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 symbol. 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 [114, 90, 103]: *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.

An example state machine is shown in Figure 4. This machine is finite in the sense that it has a finite number of states, a finite number of inputs, and a finite number of outputs (and hence also a finite number of possible transitions). This machine has 3 states, s_0, s_1, s_2, that is, S in this case is the set $\{s_0, s_1, s_2\}$. The initial state is s_0. This machine has 3 input symbols, a, p, r, which could be interpreted to mean *advance, pause, reset*, respectively. The machine has 2 output symbols and is of type Moore. State s_0 outputs the first output symbol, whereas states s_1 and s_2 output the other one. One interpretation could be that this machine models a modulo-3 counter, with s_0, s_1, s_2 representing that the counter equals $0, 1, 2$, respectively. The binary output could represent whether or not the counter is 0.

It is worth pointing out that the state machine of Figure 4 does not specify *when* exactly transitions between states are supposed to happen. Indeed, this information is not part of this state machine, and is typically not part of finite state machines in general. This is a feature, not a bug. When exactly a transition

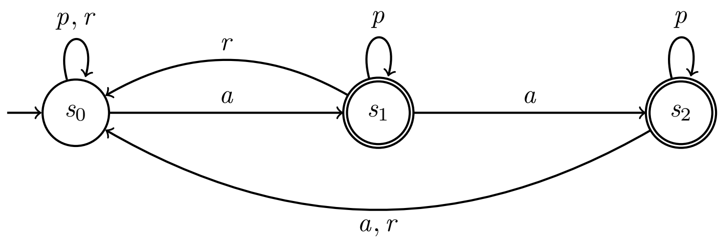

Figure 4: A finite state machine of type Moore.

occurs depends on the application. For instance, if the machine models a synchronous digital circuit, then a transition occurs at each tick of the circuit's clock. If the machine models a smart contract, and the machine's inputs model the public functions of the contract (with their possible arguments), then a transition occurs whenever a public function is called (and does not revert) – see also §8.1. When a transition occurs is often irrelevant to the logic of the machine, so it is preferable to abstract such information away.[21]

State machines are *deterministic* in the following two senses. First, given the current state and an input, the next state (also called *successor*) 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.

Deterministic state machines have the following key property: *the history of inputs completely determines the current state.* For example, if we know that the state machine receive three inputs, x_1, x_2, x_3, in that order, then we know that the current state of the machine must be

$$s''' = \delta(\ \delta(\ \delta(s_0, x_1),\ x_2),\ x_3)$$

Specifically, starting at initial state s_0, the machine first consumes x_1 and moves to state $s' = \delta(s_0, x_1)$; then it consumes x_2 and moves to $s'' = \delta(s', x_2)$; and finally it consumes x_3 and moves to $s''' = \delta(s'', x_3)$. The fact that we can reconstruct the current state from the history of inputs is a key property of deterministic state machines. It is a crucial property in many contexts, and it turns out to be important also in the context of blockchains, where the sequence of transactions completely determines the current state of the blockchain – c.f. §8.2. It is worth noting that, while histories of inputs completely determine the current state, the reverse does not hold: knowing the current state does not necessarily allow us to deduce the history of how we got here. That's because there can be many histories that lead to the same state.[22] For example, in the machine of Figure 4, the histories a, r and a, a, a both lead (back) to state s_0. And the histories a, r, a and a, p, p, p both lead to s_1.

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) [114, 90, 103]. However, the concept of a state machine is fundamental and very general. 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

[21]However, there are state-transition models such as *timed automata* [17] and *hybrid automata* [16] where time is a first-class citizen. These models are useful in applications such as embedded control systems where correct timing is extremely important for the correct behavior of the overall system [18].

[22]This is a humbling lesson in causality. We can speak of the current state of a system, but it is harder to speak of what *caused* the system to be in that state. Even in deterministic systems, there are generally many possible histories that lead to the same state; each history can be seen as a *cause* for being in that state. (The very structure of the machine, i.e., its transitions, can also be seen as a cause.) Next time you want to say something like *You made me angry* you may want to think about causality and rephrase your statement as *I am angry*. The latter statement simply reveals your current state (a fact) whereas the former assigns a cause.

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. See also §8.1.

6.4.2 Transition systems

A transition system can be seen as an *abstraction* of a state machine (or equivalently, the state machine can be seen as a *refinement* of the transition system). The transition system is an abstraction in the sense that it omits information about the inputs. As a result, transitions become *non-deterministic*, that is, a state can have multiple successor states. For example, consider the labeled state-transition model shown in Figure 5. If we ignore the labels on its transitions, the model is a non-deterministic transition system: e.g., from state 0 the system can move either to state 1 or to state 2. If, on the other hand, we take the labels `TRY1`, `TRY2`, and so on, to represent input symbols, then we can look at this model as *almost* a state machine (why "almost"?).

Formally, a transition system is a tuple (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 (also called *transition relation*). Note that, contrary to a state machine where s_0 is an *element* of S (meaning there is a unique initial state), here S_0 is a *subset* of S, meaning there can be many initial states. Which state does the system start in? Some state in S_0, chosen non-deterministically. This can be used to model, for example, cases where we don't know how some state variables might be initialized.

The set of transitions T is a subset of $S \times S$: this says that transitions are pairs of states. Such a pair $(s, s') \in T$ represents a transition from state s to state s'. Because we allow T to be any subset of $S \times S$, we can have cases where say, both (s, s') and (s, s'') are elements of T: this models the fact that the next state from s is chosen non-deterministically. We can also have the case where T contains *no* pair (s, s'), for any s': this models the fact that s is a *deadlock*, i.e., if the system arrives at state s, it cannot move at all. Does the system of Figure 5 have deadlocks?

Transition systems look deceptively simple, but they are a very general model which can capture pretty much all dynamical systems.[23] Transition systems are as fundamental to computer science as numbers are to mathematics and physics. For a more in-depth discussion of transition systems see, for instance, [109, 24, 128]. (Additional material will be provided on canvas.)

6.4.3 Symbolic representations of state machines and transition systems: next-state variables

Graphical representations of state-transition systems like the ones in Figures 4 or 5 can work when the model is small. But because of state explosion, real-world system models are huge. So we cannot draw those models graphically, and we cannot even write down all their states and transitions one by one. Instead, we use *symbolic* representations. There's nothing frightening about the word *symbolic*. All it means is that we define things in terms of *state variables* rather than *states*. This is anyway what (imperative programming language) programmers always do: they write programs that define how program variables (i.e., state variables) are updated during the execution of the program. An assignment statement `x := x+1` (or in the bad C syntax `x = x+1`) says that if the value of state variable `x` in the current state is n, then it should be $n + 1$ in the next state.

Assignment statements like `x := x+1` work fine for representing symbolically the transition functions of deterministic systems. But how to represent symbolically the transition relation of a non-deterministic transition system? A cool trick is to use *next-state variables*. A next-state variable is a variable name (a symbol) that represents the value of a state variable at the next state. Typically, we obtain the next-state variable of a state variable x by adding a prime $'$, as in x'. So if `x` and `y` are the state variables of our

[23] with the exception of stochastic systems, which require probabilistic transition systems, i.e., transition system with probabilities on their transitions. Even in the non-stochastic case, transition systems sometimes need to be extended with additional *fairness* assumptions. These are needed in order to prove *liveness* properties. See §6.12.

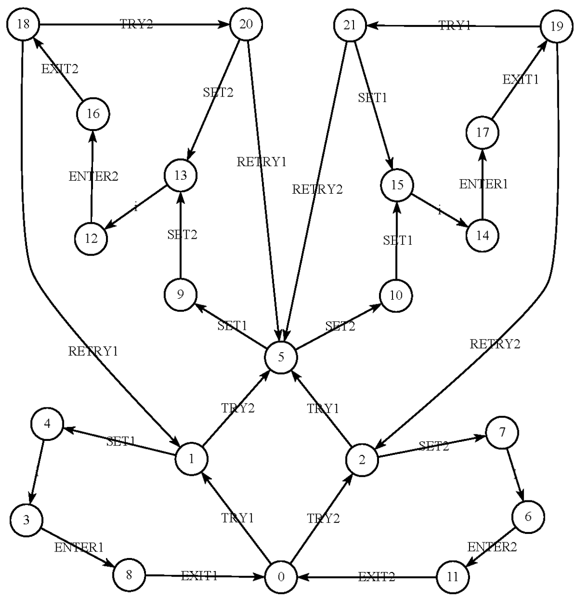

Figure 5: Example of a (labeled) transition system.

program, `x'` and `y'` are the corresponding next-state variables. But note that using $'$ is just notation. We could have chosen a different notation, e.g., `next(x)` (that's what the nuXmv model-checker uses [42]).

The cool thing about next-state variables is that they can seamlessly handle deterministic as well as non-deterministic systems. For example, in the case of a deterministic update, we can write `x' = x+1` instead of `x := x+1`. But we can also write something like `x' = x+1 || x' = x-1` which denotes a non-deterministic transition relation where `x` is either incremented or decremented by 1 at each step. In this example, `x' = x+1 || x' = x-1` is a Boolean expression (or generally a *predicate*) on the current and next-state variables, and `||` represents logical disjunction. Using non-ASCII logical symbols we could have written it as $x' = x + 1 \vee x' = x - 1$.

A **symbolic transition system** is a transition system represented in a symbolic way.[24] Formally, a symbolic transition system is a triple $(X, \mathsf{Init}, \mathsf{Next})$, where X is a set of (typed) state variables, Init is a logical formula over X, and Next is a logical formula over $X \cup X'$, where X' is the set of next-state variables corresponding to X. For instance, if $X = \{x, y, z\}$ then $X' = \{x', y', z'\}$. Init defines the set of initial states: a state is initial iff (if and only iff) it satisfies Init. For example, if Init is the formula $x = 0 \wedge y \geq z$ then the set of initial states contains all states (and only those states) where state variable x equals zero, and the value of state variable y is greater or equal to the value of state variable z. For instance, $(x = 0, y = 0, z = 0)$ is an initial state because it satisfies Init; but $(x = 0, y = 0, z = 1)$ violates Init and therefore is not an initial state. Next defines the transition relation: a transition from state s to state s' is valid if the pair (s, s') satisfies Next. What does that mean exactly? It means that if we replace all current-state variables in Next by their corresponding values in s, and all next-state variables by their values in s', we get a formula that evaluates

[24]Symbolic representations are based on the beautiful correspondence between logic and set theory, where conjunction, disjunction and negation correspond to set intersection, union, and complementation, respectively. For example, the formula $x > 0$ can be interpreted as the set of all x greater than 0, the formula $x + y < 10$ as the set of all pairs (x, y) such that $x + y$ is less than 10, and so on. Then, the formula $x > 0 \wedge x < 5$ can be seen as the intersection of the "sets" $x > 0$ and $x < 5$. Interestingly, logical implication can be interpreted in two ways: $A \rightarrow B$ can either be interpreted as the set $\overline{A} \cup B$, where $\cup$ denotes set union and $\overline{A}$ denotes the complement of A; or $A \rightarrow B$ can be interpreted as the proposition $A \subseteq B$, which states that A is a subset of B. (That's why you will sometimes see texts that use $\subset$ as the symbol for logical implication, $\cup$ for disjunction, and $\cap$ for conjunction.)

```
$\textsc{constant}\ \mathit{Server},\ \mathit{Client}$
$\textsc{variable}\ \mathit{locked},\ \mathit{held}$

$\mathit{Init}\ \triangleq$
    $\land\ \mathit{locked} = [i \in \mathit{Server} \mapsto \textsc{true}]$
    $\land\ \mathit{held} = [i \in \mathit{Client} \mapsto \{\}]$

$\mathit{Connect}(c,\ s)\ \triangleq$
    $\land\ \mathit{locked}[s] = \textsc{true}$
    $\land\ \mathit{held}' = [\mathit{held}\ \textsc{except}\ ![c] = \mathit{held}[c] \cup \{s\}]$
    $\land\ \mathit{locked}' = [\mathit{locked}\ \textsc{except}\ ![s] = \textsc{false}]$

$\mathit{Disconnect}(c,\ s)\ \triangleq$
    $\land\ s \in \mathit{held}[c]$
    $\land\ \mathit{held}' = [\mathit{held}\ \textsc{except}\ ![c] = \mathit{held}[c] \setminus \{s\}]$
    $\land\ \mathit{locked}' = [\mathit{locked}\ \textsc{except}\ ![s] = \textsc{true}]$

$\mathit{Next}\ \triangleq$
    $\lor\ \exists\, c \in \mathit{Client},\ s \in \mathit{Server} : \mathit{Connect}(c,\ s)$
    $\lor\ \exists\, c \in \mathit{Client},\ s \in \mathit{Server} : \mathit{Disconnect}(c,\ s)$

$\mathit{Spec}\ \triangleq\ \mathit{Init} \land \Box[\mathit{Next}]_{\langle \mathit{locked},\ \mathit{held} \rangle}$

$\mathit{Safe}\ \triangleq\ \forall\, c_i,\ c_j \in \mathit{Client} : (\mathit{held}[c_i] \cap \mathit{held}[c_j] \neq \{\}) \Rightarrow (c_i = c_j)$
```

Figure 6: A simple parameterized protocol defined in TLA+ – taken from [147].

to true. For example, let Next be the formula $x' = x + 1 \lor x' = x - 1$, and let $s_0 = (x = 0, y = 0, z = 0)$, $s_1 = (x = 1, y = 0, z = 0)$ and $s_2 = (x = 2, y = 100, z = 42)$ be three states. Then (s_0, s_1) is a valid transition, and (s_1, s_2) is also a valid transition (why?). But (s_0, s_2) is not a valid transition (why?).

6.4.4 TLA+

There are many practical languages for writing symbolic transition systems. In this course we will use the TLA+ language [101, 97]. TLA+ is used both in academia as well as in the industry / by practitioners [28, 116, 92, 79, 153]. A TLA+ model, e.g. the one shown in Figure 6, is a symbolic representation of a transition system using the primed next-state variable approach. You will learn to read and understand such models.

It is very important to internalize the distinction between state and state variable, and also use these two terms correctly, consistently. A *state variable* is just a name or a symbol. A *state* is obtained by giving values to the state variables. This is true both for Solidity contracts, as well as formal modeling languages like TLA+. In a Solidity contract with state variable `total` of type `uint8`, `total` is the state variable. This variable can be in any of 2^8 possible states (why?). In the TLA+ model of Figure 6, *held* and *locked* are state variables. What are some possible states of this model?

6.4.5 Syntax vs semantics

Another crucial distinction which you need to internalize is the distinct concepts of *syntax* and *semantics*. Syntax refers to the symbols that represent something, e.g., the ASCII text of your program. Semantics refers to the mathematical object that tries to capture the meaning of the syntax. In our case, the semantics will often be some kind of transition system.

Formal languages like TLA+ have precise rules that allow to mathematically define the semantics from the syntax. What this means is that every TLA+ program (syntax) has a clearly defined transition system (semantics) associated with it. In fact, the TLA+ toolset allows to generate and draw this transition system automatically. For instance, consider the TLA+ specification [25] of Figure 6. The reachable state-transition

[25] Lamport and the TLA+ community use the term *specification* to refer to both the formal model of the implementation (i.e.,

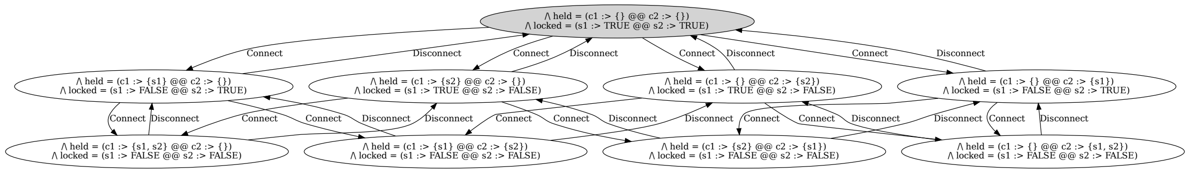
/\ held = (c1 :> {} @@ c2 :> {})
/\ locked = (s1 :> TRUE @@ s2 :> TRUE)
/\ held = (c1 :> {s1} @@ c2 :> {})
/\ locked = (s1 :> FALSE @@ s2 :> TRUE)
/\ held = (c1 :> {s2} @@ c2 :> {})
/\ locked = (s1 :> TRUE @@ s2 :> FALSE)
/\ held = (c1 :> {} @@ c2 :> {s2})
/\ locked = (s1 :> TRUE @@ s2 :> FALSE)
/\ held = (c1 :> {} @@ c2 :> {s1})
/\ locked = (s1 :> FALSE @@ s2 :> TRUE)
/\ held = (c1 :> {s1, s2} @@ c2 :> {})
/\ locked = (s1 :> FALSE @@ s2 :> FALSE)
/\ held = (c1 :> {s1} @@ c2 :> {s2})
/\ locked = (s1 :> FALSE @@ s2 :> FALSE)
/\ held = (c1 :> {s2} @@ c2 :> {s1})
/\ locked = (s1 :> FALSE @@ s2 :> FALSE)
/\ held = (c1 :> {} @@ c2 :> {s1, s2})
/\ locked = (s1 :> FALSE @@ s2 :> FALSE)
Connect
Disconnect

Figure 7: The reachable state-transition graph of the TLA+ model of Figure 6 for the case where `Server={s1,s2}` and `Client={c1,c2}`.

graph of this system is shown in Figure 7, for the case where there are two servers and two clients. Specifically, parameter *Server* is instantiated to the set {`s1,s2`} and parameter *Client* is instantiated to the set {`c1,c2`}. Figure 7 has been automatically generated with the following commands:

```
java -jar tla2tools.jar -dump dot,actionlabels graph.dot lockserverfmcad22.tla
dot -Tpdf graph.dot > graph.pdf
```

6.4.6 Nondeterminism

FSMs are typically deterministic, whereas transition systems are typically nondeterministic. Where does the nondeterminsm come from? We have already seen examples illustrating the possible sources of nondeterminism.

One possible source is *abstraction*. For example, consider the transition system of Figure 5. If we include the transition labels in this system, and consider these labels as input symbols, then the system can be considered to be a deterministic FSM (without outputs, or simply where the state is the output). The system with labels is deterministic, because given a state and a label (input) the next state is uniquely determined. (This system is also incomplete, because only the reaction to *some* inputs is defined at every state.) On the other hand, if we omit the transition labels, then the system becomes nondeterministic (because, for instance, state 0 has two possible successors, and so does state 5 and state 20). Omitting the transition labels can be seen as discarding some (perhaps irrelevant) information about the system, which is what abstraction means.

Another possible source of nondeterminism is *concurrency*. Consider again the transition system of Figure 5. We can think of this system as capturing the behavior of two concurrent processes which each try to acquire access to a *critical section*. Each process is sequential and deterministic (for example, process 1 can be seen as performing the sequence of events TRY1, SET1, i (for *internal* action), ENTER1, and EXIT1, repeatedly in a loop), but when both processes execute concurrently, their behaviors *interleave*, resulting in the system of Figure 5. Another example of concurrency resulting in nondeterminism is the TLA+ program of Figure 6, with its corresponding nondeterministic transition system (Figure 7).

A third source of nondeterminism comes from modeling *open* systems. An open system is a system that receives inputs from its environment. For example, the contract `SimpleStorage` of Figure 2 is an open system because the environment has to choose (1) which of the two functions to call, `set` or `get`, and (2) what value to pass to `set` if it decides to call `set`. On the other hand, a monolithic transition system is a *closed* system; it has no inputs, and in fact it has no distinction of inputs and outputs.[26] Verification languages typically allow to describe systems in a modular way, as a collection of interacting subsystems. The subsystems may be open, but the overall system must be closed, so that a monolithic transition system can be generated

of the system that we want to verify) as well as the formal specification of its correctness properties. So, when you see people talking about TLA+ specifications, they mean not just the correctness specification ϕ, but also the model P of the system – c.f. §6.3.

[26]Compositional frameworks like [22, 131] do distinguish between inputs and outputs, but we don't discuss them here. On the other hand, we will discuss synthesis, which also distinguishes between inputs and outputs – c.f. §6.15.

from it. Nondeterminism comes handy in such cases. You will explore the use of nondeterminism for that purpose in your assignments.

6.4.7 PlusCal

PlusCal is an imperative language which can be automatically compiled into TLA+. Leslie Lamport, the creator of PlusCal, calls it an "algorithm language" [98]. Why bother with PlusCal? Why learn yet another language? Because it is easier to model programs like smart contracts in PlusCal than in TLA+. For instance, assignment sequences, while loops, and function calls are all directly available in PlusCal but not in TLA+. See also Lamport's discussion in [98].

Regarding running PlusCal, here's the commands I'm using:

```
  cp pcaltest01.pcal toto.tla; pcal -nocfg toto; mv toto.tla test.tla
  tlc test.tla
```

where `pcaltest01.pcal` is the source PlusCal file, and `pcal` and `tlc` are aliases:

```
alias pcal='java -XX:+IgnoreUnrecognizedVMOptions -XX:+UseParallelGC -cp $TLA_HOME/tla2tools.jar pcal.trans '
alias tlc='java -XX:+IgnoreUnrecognizedVMOptions -XX:+UseParallelGC -cp $TLA_HOME/tla2tools.jar tlc2.TLC '
```

You can simulate PlusCal programs by translating them first into TLA+, and then using https://will62794.github.io/spectacle. Note that it's the generated TLA+ that gets simulated, not the original `.pcal` file: so if you're using the command above, make sure you load the `test.tla` file into the website.

To illustrate how we can use PlusCal to model smart contracts, let us write a PlusCal program capturing the `SimpleStorage` contract of Figure 2. The corresponding PlusCal program is shown in Figure 8. Make sure you understand this code (be ready to discuss it in class). How many nondeterministic statements does this code contain? Also make sure you translate this PlusCal program into TLA+ and study the automatically generated TLA+: do you understand which PlusCal statements the symbolic transitions (actions) of the generated TLA+ program correspond to? Finally, generate the state-transition graph PDF and/or simulate the TLA+ program and make sure you understand its entire behavior (states and transitions).

6.5 Model-checking

In a nutshell, the model-checking problem is to check, for given system model P and correctness specification ϕ, whether $P \models \phi$, i.e., whether P satisfies ϕ – c.f. §6.3. There are many variants of the model-checking problem, depending on what formalism P and ϕ are expressed in. Typically, P is some kind of transition system, and ϕ is a formula in some logic. A lot of work in model-checking has been done for variants of the problem where ϕ is a formula of some kind of *temporal logic* [129, 96, 109, 24]. In fact, the acronym TLA+ comes from *Temporal Logic of Actions* [101]. We will get a glimpse of temporal logic in §6.13.

6.5.1 State assertions

For now, we focus on what is perhaps the simplest variant of the model-checking problem where: P is a transition system, typically given in symbolic form, e.g., a TLA+ program like the one of Figure 6; ϕ is an *assertion* about the states of the system. Such an assertion is typically a Boolean or more general *state* formula (also called *state predicate*), that is, a formula that refers to state variables. Importantly, state formulas can be evaluated simply by knowing the current state: we don't need to know past or future states (for temporal logic formulas, we do).

An example of such a ϕ is the predicate *Safe* in Figure 6. In this system, the state variables are *held* and *locked*. (*Server* and *Client* are *parameters* which are fixed in the beginning and don't change afterwards, so we don't consider them to be state variables.) Predicate *Safe* only refers to state variable *held*; *held* maps clients to sets of servers. Suppose there are two clients, c_1, c_2, and two servers, s_1, s_2. Let q_1 be a state where $held[c_1] = \{s_1, s_2\}$ and $held[c_2] = \{s_2\}$. Then q_1 does not satisfy *Safe*, meaning that *Safe* evaluates to false on q_1 (why?). On the other hand, let q_2 be a state where $held[c_1] = \{s_1\}$ and $held[c_2] = \{s_2\}$. Then q_2 satisfies *Safe* (why?).

```
---- MODULE test ----
EXTENDS Naturals, TLC, Sequences

CONSTANT Address   \* a set like {a1,a2}
CONSTANT MaxValue  \* a small number, e.g., 2

(* --algorithm transfer {
   variables storedData = 0;
             returnedValue = 0 ;

   procedure set(msg_sender, x)
   { Lset:  storedData := x;
            return
   }

   procedure get(msg_sender)
   { Lget:  returnedValue := storedData;
            return
   }

   { L1: with (sender \in Address; z \in 0..MaxValue)
            either { call set(sender, z) }
            or { call get(sender) };
            or skip;
     L2: goto L1
   }
}
end algorithm; *)
====
```

Figure 8: PlusCal program capturing the `SimpleStorage` contract of Figure 2. The `msg_sender` arguments are unused in the functions but are included to explain how you might model Solidity's `msg.sender`.

6.5.2 Reachable states

We now know what it means for a state q to satisfy a state assertion ϕ: ϕ evaluates to true at state q. But what does it mean for a transition system P to satisfy a state assertion ϕ? It means that every *reachable* state of P satisfies ϕ. A reachable state is a state that can be generated by starting from some initial state and following the transitions of the system, some number of times (including zero times, i.e., all initial states are by definition reachable).

You should understand at this point what an initial state of a TLA+ program like the one in Figure 6 is, and also what a transition is in such a system (if you don't, see §6.4.3, §6.4.4, §6.4.5, and the TLA+ references [101, 97]). But knowing these things does not immediately help us answer the model-checking question: does P satisfy ϕ? For example, does the transition system represented by the TLA+ program of Figure 6 satisfy the *Safe* predicate? By looking at the TLA+ code, the answer to this question is not obvious. In fact, it's not obvious what the reachable states even are.

6.5.3 Explicit-state model-checking – reachability analysis

Explicit-state model-checking (also called *enumerative* model-checking or *reachability analysis*) attempts to answer the above question automatically, by generating all reachable states of the system, one by one. This is what TLC essentially does: TLC is the explicit-state model-checker of TLA+ – https://lamport.azurewebsites.net/tla/tools.html. This is also what Holzmann's pioneering model-checker Spin does [84, 86]. These tools treat transition systems essentially as graphs, and employ algorithms like depth-first search (DFS) or breadth-first search (BFS) to generate all states in the graph (i.e., all reachable states). For every generated state q, the tool evaluates the state assertion ϕ on q. If ϕ evaluates to true, the search continues; otherwise, the tool has discovered a violation of ϕ. At this point, the tool can also typically generate a counterexample: a path from an initial state all the way up to q. This is a useful debugging mechanism as it helps understand the steps that lead to the violation.

You can look at Figure 7 and try to check "by hand" whether *Safe* is satisfied in this system (you may have to magnify the PDF to be able to read each state).

6.5.4 Infinite-state systems and undecidability

Unfortunately, generating the set of all reachable states is not always possible. To begin with, there are cases where there is an infinite number of reachable states, so we cannot generate all of them (we can still try to generate as many as we can in the hope of catching bugs; but we will not be able to prove the absence of bugs). For instance, continuous-time models such as timed or hybrid automata have infinite state spaces [17, 16]. Discrete systems can also be infinite, e.g., imagine a program manipulating lists of unbounded length.

For most infinite-state models, the "simple" model-checking problem of checking state assertions is *undecidable*, meaning there is no algorithm that is guaranteed to always terminate and give a correct yes/no answer. This follows from Turing's celebrated result on the undecidability of the halting problem for Turing machines [165, 104]. You have probably seen this result in your theory of computation class (if you haven't you are encouraged to take such a class; in the meantime, you can read this 1-page document for the intuition [162]). In fact, the problem is undecidable even for very limited models, such as *two-counter machines*: these are finite-state automata with two integer counters that can only be incremented by 1, decremented by 1, or tested for zero.

One might object that all physical computers have finite memory, and any software running on a computer is bounded by the memory of that computer and therefore can only be finite-state; so we don't have to worry about undecidability, right? This view is short-sighted for a number of reasons. First, in many cases the correctness of a program is independent from memory considerations. For example, we would like to prove the correctness of a distributed protocol for any number of nodes. And we would like to prove the correctness of a program that sorts lists for *any* input list, no matter how long this list is. This is much better than proving correctness only for lists of length up to some bound N, because our result is *timeless*: we know that our sorting program will work correctly on any computer, no matter how much memory that computer has, provided it has enough memory for the program to run to completion. If we only proved correctness for bounded memory, then we would have to re-do the proof every time computers gain more memory than our bounds. Second, finiteness may imply decidability, but it does not solve the problem of *state explosion* – §6.5.5. Naively enumerating all states will not get us far if the number of states is astronomical. Third, and perhaps paradoxically, it is sometimes *easier* to abstract away from finiteness considerations and to prove things in the unbounded case. We will indeed do that several times in this class. For instance, when we prove ♠ correctness of insertion sort in Lean (§6.6) we will prove it for any input list, not just for lists of bounded length.[27] Also, when we verify Hoare triples (§6.10) using an SMT solver like Z3 (§6.8), we will see that we can prove things about programs that manipulate, e.g., unbounded integers.

We should remark that undecidability hasn't stopped researchers from developing automated or semi-automated verification tools for highly expressive languages (TLA+ is such a language) and even for real-world programming languages. SV-COMP, an annual competition on software verification, features a number of such tools – `https://sv-comp.sosy-lab.org/`. You might wonder how is it possible to develop a tool that solves an undecidable problem. Isn't this by definition impossible? Not quite. It is impossible to build a tool that is guaranteed to *always terminate and give the right answer* to an undecidable problem. But it is OK for such tools to sometimes not terminate, or to terminate by reporting *don't know*.

6.5.5 State explosion

Even if we restrict ourselves to finite-state systems (for which reachability is trivially decidable – why?) our problems are not over, because state-transition models can quickly get very big. This is known as the *state-explosion problem* and is one of the main obstacles in formal verification. Just how big can these models get? Imagine a digital circuit with N boolean transistors (each transistor can be in one of two states, either 0 or 1): this system has 2^N possible states. How big is N? Modern circuits can have billions of transistors. Two to the billion is an unthinkably big number. Software models can also grow huge. How many states does a Solidity contract with a single `uint256` state variable have? What if that contract also has also a state variable of type `mapping(address => uint256)`?

You may (correctly) observe that these numbers count the set of *all possible* states, but not all of these states maybe *reachable*. For example, it may be that a contract with two state variables x, y both of type `uint256` ensures that $x \leq y$ always holds. Then many states are impossible, e.g., $(x = 10, y = 9)$, $(x = 10, y = 8)$, $(x = 11, y = 10)$, and so on. Still, even if only a small fraction of the potential set of all states happens to be reachable, the numbers are so large that even this small fraction is huge.

For example, consider again the TLA+ specification of Figure 6. This model is *parameterized* by the sets *Server* and *Client*. Figure 7 shows the reachable state-transition graph of this model for two servers and two clients. What if we increase the number of servers and clients? With six servers and six clients, the reachable state graph has 117649 states. These were generated in about 17 secs on my laptop, using the TLA+ model-checker TLC – `https://lamport.azurewebsites.net/tla/tools.html`. You should remark that a $3\times$ increase in the number of servers and clients resulted in a $13000\times$ increase in the number of reachable states. That's an illustration of state-explosion.

6.5.6 Techniques against state explosion

Over the years, formal methods researchers developed a large number of very clever techniques to tackle state explosion. We will not discuss any of these techniques in depth in this course. But let us mention here some of the techniques implemented in tools such as TLC and Spin. Specifically, TLC uses parallelization which exploits, say, the multiple cores in your computer to explore the state space in parallel. TLC also uses *symmetry reduction* to reduce the part of the state space that needs to be explored [46]. Spin uses techniques such as *partial-order reduction* [166, 75] to reduce the part of the state-space that needs to be explored, and *bitstate hashing* [85] to store the reachable states efficiently.

Another important technique is *symbolic model-checking* with BDDs (Binary Decision Diagrams) [37]. BDDs are an efficient data structure for representing Boolean formulas. The idea of symbolic model-checking

[27]When you finish that proof, you may try to re-do it for bounded lists. You will probably find out that both the claims and the proofs become more complicated.

is to represent the set of reachable states as one BDD, and instead of enumerating states one by one, to compute the set of one-step successors of a given set of states (represented as a BDD) in one shot. A number of model-checkers use BDDs, including the model-checker nuXmv [42].

Although the above techniques can often result in impressive gains, they do not completely solve the state-explosion problem.

6.6 Theorem proving with LEAN

In the model-checking approach we view programs as state-transition systems (this is also called *operational semantics*). This approach is very general, and applies not just to software, but to all kinds of dynamical systems, e.g., embedded or so-called *cyber-physical systems* (systems combining discrete and continuous dynamics, like timed automata [17] or hybrid automata [16]).

Theorem-proving is a complementary approach, where both programs and their correctness specifications are expressed as mathematical statements (claims) in some logical framework, and verification consists in proving these claims (as automatically as possible). The tools that implement this approach are called *theorem provers* or *proof assistants*. The latter term emphasizes the fact that these tools are typically interactive. Rather than proving theorems fully automatically, the proof assistant keeps track of the proof and does all the necessary book-keeping, making sure, for instance, that all proof steps are mathematically (logically) valid, and no steps have been forgotten. The proof itself is written by the (typically human) user, who has to discover all the "creative" aspects of the proof (you will see examples of that in class and in assignments). In this course, we will use Lean [53, 6]. (I also use Lean in my *CS 2800: Logic and Computation* class [163].) Other well-known proof assistants include Isabelle [117, 5] and Rocq (was: Coq) [29, 8]. (There are many others: see [163] for more references.)

You can think of Lean as being to proofs-by-hand what programming languages are to pseudocode. You can write pseudocode on a piece of paper, and this is often better for communicating ideas than writing real code. But you cannot execute pseudocode. For that, you need a real program, say, in C or Java or Python. Writing real code also requires you to iron the details which are missing from your pseudocode. During this process, you might realize that your pseudocode omits important aspects, or is even buggy. The same is true with Lean. You can do a sketch of proof, or even a full proof by hand, but you cannot "execute" it. Lean "executes" the proofs that you write, in the sense of checking them and making sure they are sound and complete.

Concretely, Lean is a functional programming language, coupled with a formal logic and a theorem prover. In Lean programs are viewed as mathematical functions (this is related to so-called *denotational semantics*, which we will not worry about here). In fact, in Lean, theorems are also functions (!) : they are functions that return proofs! This is a fascinating aspect of Lean's formal logic, which is called *dependent type theory* and which has its origins in the very foundations of mathematics [127]. You can think of this theory as a very powerful logic. It is a higher-order logic meaning that you can quantify over everything, including functions, types, etc. In this class, we will not go into the theoretical aspects of logic and types. Instead, we will gain hands-on experience and through it understand statements like *theorems are functions that return proofs*. For in-depth discussions of logic and types see [80, 127]. A hands-on experience of logic and theorem proving from a software verification perspective, and using the Coq proof assistant, is provided in [9].

In this class, we will learn Lean "by doing". We will see Lean in action during class (all code will be provided on canvas). Then you will do homeworks in Lean, to experience what theorem proving is about. The code given in Figure 9 is a simple implementation of insertion sort in Lean. In your assignments you will investigate questions like *what does it mean for this code to be correct?* and *how do we formally prove correctness?* This will teach us many valueable lessons about theorem proving, but also about formal verification as a whole.

Theorem-proving approaches to verification, such as Lean's, are very powerful and general, but also difficult to automate. This follows from fundamental undecidability results in logic (e.g., see [80]). Having said that, automated tools like *satisfiability solvers* are making progress and are becoming instrumental to formal verification. We will discuss this in the sections that follow.

```
-- is one nat less or equal than another nat?
def leq : Nat -> Nat -> Bool
  | 0, _ => true
  | Nat.succ _, 0 => false
  | Nat.succ x, Nat.succ y => leq x y

-- insert a nat to the right place in a list
def insrt : Nat -> List Nat -> List Nat
  | x, [] => [x]
  | x, (y :: L) =>  if (leq x y)
                    then x :: (y :: L)
                    else y :: (insrt x L)

-- insertion sort
def isort : List Nat -> List Nat
  | [] => []
  | (x :: L) => insrt x (isort L)
```

Figure 9: Insertion sort written in the Lean functional programming language (Lean 4).

Frameworks like Lean are complementary to frameworks like TLA+, not just in terms of the degree of verification automation that they provide, but also and foremost because their languages are more or less suited to different domains. For instance, systems like smart contracts have concepts like states, transitions, and concurrency. These are not first-class citizens in a functional programming language like Lean's, but are first-class citizens in TLA+. Although these concepts could be encoded in Lean, doing so would result in more complex and somewhat less natural models. On the other hand, Lean's logic is much more powerful than TLA+'s logic, and Lean has an extensive mathematical library which TLA+ lacks. In fact, Lean is actively used in the formalization of advanced mathematics, e.g., see `https://leanprover-community.github.io/blog/posts/lte-final/` and `https://terrytao.wordpress.com/2023/11/18/formalizing-the-proof-of-pfr-in-lean4-using-blueprint-a-short-tour/`. Lean is also used in domains related to blockchains. For instance, see the zkEVM Formal Verification Project by the Ethereum Foundation: `https://verified-zkevm.org/`; or cLean [58]; or RISC Zero zkVM formalisation – `https://www.nethermind.io/blog/towards-formal-verification-of-the-first-risc-v-zkvm` and `https://github.com/NethermindEth/risczero-fv/`.

6.7 Theorem-proving: lessons learned

It is worth pausing and reflecting on our experience so far with theorem-proving. We learned some important lessons (many of which also apply to formal verification in general).

6.7.1 Expressing correctness precisely is not always trivial

The first lesson is that, even for this relatively simple example (list sorting) it took us some time to figure out what exactly correctness means, and how to express it formally. Moreover, writing the formal spec involved writing more Lean code (functions like `isSorted` and `permutation`). And, as we noticed, these functions may themselves have bugs, which would mean that our spec is incorrect! Also, we might forget a property altogether, in which case our spec would be incomplete.

The second lesson is that for this small program of 12 lines-of-code (LoC) in total (counting the functions `isort`, `insrt`, and `leq`, without comments, Figure 9) we had to write 10 to 18 LoC of formal spec (counting the functions `isSorted` and `permutation`, and depending how exactly these are written). So the size of the specification is as large as the size of the program, if not larger!

6.7.2 Incomplete or inconsistent specifications

A third lesson about specifications is that we might sometimes forget some requirements. For example, we might specify that `isort` must return sorted lists, but we might forget that the output list should also be a permutation of the input. In such cases, our specification is *incomplete*.

In other cases, our specification might be *inconsistent*, i.e., logically self-contradictory, which means that we assert something and also its negation.

6.7.3 Formal proofs are non-trivial

A fourth lesson is that proving formally correctness using a proof assistant like Lean is non-trivial. Not only the size of the proof itself is large (about 600 LoC!) but it also took us a long time to write the proof. Although the process was at times tedious and painful, it was also rewarding and highly educational. Apart from the reward of completing the proof itself (equivalent to an infinite number of tests, since the proof holds for all possible input lists!), we had to come up with many crucial lemmas in the process. These lemmas shed light to the key properties of helper functions like `insrt` that make our program correct (e.g., the fact that `insrt` preserves sortedness). Other crucial lemmas express key properties of lists (e.g., if a non-empty list is sorted, its tail is also sorted).

6.7.4 The how, what, and why of programming

These are fundamental lessons about formal methods and verification in general. We can look at a program as telling us *how* to do *something*, but not really telling us *what* that "something" is – indeed, this is often written in the comments, since it is absent from the code itself. A formal specification defines that "something": it makes precise *what* the program is supposed to do, namely, what properties it's supposed to have. A proof reveals *why* the program actually has those properties, i.e., why it works, and why we should trust that it works.[28] Both the *what* and the *why* are typically absent from the code of the program itself. It is therefore no wonder that we need several more LoC to add this information. There's no free lunch!

6.8 SAT and SMT solvers

Our experience with theorem proving (§6.6, §6.7) shows that proving theorems formally is non-trivial. Wouldn't it be nice if we had tools that can formally prove (or disprove) theorems fully automatically? This has indeed been a longstanding dream of both mathematics and computer science.[29] Unfortunately, fundamental undecidability and incompleteness results in logic (e.g., see [80]) imply that this dream cannot be achieved in general.[30] Despite this, researchers have been making steady progress in developing algorithms (*decision procedures* [34, 94]) for several fragments of logic which are important in practice.

The most important (and basic) fragment is Boolean logic: Boolean formulas where all variables are of type Boolean. The *satisfiability (SAT) problem* is to check whether such a formula is satisfiable, that is, whether there exists an assignment of true/false values to all variables, which makes the formula evaluate to true. The SAT problem is perhaps the most famous NP-complete problem [104]. Although it is easy to see why SAT is decidable (why?), the SAT problem has been considered intractable until recently. This has changed with the impressive advances of SAT solvers since the early 2000s [107, 106]. Modern SAT

[28] A proof can therefore be seen as an *explanation* of why the program works. Contrast this to the current discussion on explainability of AI.

[29] Automated theorem proving motivated Turing's seminal paper [165] (see footnote that follows) so one might say that this dream gave birth to computer science itself.

[30] In the beginning of the 1900s mathematics faced a crisis of foundations, triggered by things like Russell's paradox. This crisis motivated a number of further developments, many of which turned out to be fundamental for both modern logic and computer science. Bertrand Russell's *Principia Mathematica* laid the foundations for modern type theory (the logic of Lean!). David Hilber's program aspired to formalize all mathematics and to even devise algorithms for deciding the truth or falsity of any mathematical statement. Goedel's incompleteness theorems and Turing's undecidability result showed that this program cannot be achieved in general (the *Entscheidungsproblem* in Turing's paper [165] refers to Hilbert's *decision problem*). A beautiful account of this fascinating history of logic is given in [63]. For an in-depth discussion, see Harrison's book [80].

solvers can solve formulas with hundreds of thousands of variables, c.f. the SAT Competition – https://satcompetition.github.io/.

SMT (SAT Modulo Theories) solvers go beyond SAT and can handle more general formulas, with variables representing integers, arrays, functions, and so on – see [27] and https://smt-lib.org/. The decidability/hardness of handling such formulas depends on the type of variables and the underlying logical theory [34, 80, 27]. Like SAT solvers, SMT solvers have also been making significant advances in recent years, c.f. the SMT Competition – https://smt-comp.github.io/. This is partly due to the fact that SMT solvers make internal calls to SAT solvers (c.f. slide 8 of [27]).

6.9 Bounded model-checking

The advances in SAT and SMT solvers have had a big impact on formal verification. They have enabled the development of several methods which reduce the verification problem to a satisfiability problem. For instance, *bounded model-checking* (BMC) [32] consists in *unrolling* the transition relation a finite number of times, thus encoding bounded (finite-horizon) reachability as a satisfiability problem and using a SAT/SMT solver to solve the latter. A notable bounded model-checker is CBMC which works on C and C++ programs – https://www.cprover.org/cbmc/.

Let us briefly explain BMC. Consider a symbolic transition system $(X, \mathsf{Init}, \mathsf{Next})$. Suppose we are also given a formula Bad over X; Bad represents a set of "bad" (unsafe, erroneous, ...) states. Let $X_0, X_1, ...$ be copies of the set of state variables X, where in X_i all variables in X are indexed by i. For example, if $X = \{x, y, z\}$ then $X_3 = \{x_3, y_3, z_3\}$. Also let $\mathsf{Init}[X_0]$ denote Init with X variables renamed into X_0 variables; similarly, let $\mathsf{Bad}[X_n]$ denote Bad with X variables renamed into X_n variables; and let $\mathsf{Next}[X_i, X_{i+1}]$ denote Next with X and X' variables renamed into X_i and X_{i+1} variables, respectively. Then consider the following formula, for given $n \geq 0$:

$$\mathsf{Init}[X_0] \wedge \mathsf{Next}[X_0, X_1] \wedge \mathsf{Next}[X_1, X_2] \wedge \cdots \wedge \mathsf{Next}[X_{n-1}, X_n] \wedge \mathsf{Bad}[X_n] \qquad (1)$$

If formula (1) is satisfiable, it means we can find values for all variables in $X_0, X_1, ..., X_n$ to make the formula true. But that means that we can find $n+1$ states, $s_0, s_1, ..., s_n$, where s_i is the assignment of values to X_i, such that: (1) s_0 is an initial state; (2) there is a valid transition from each s_i to s_{i+1}, for $i = 0, ..., n-1$; and (3) s_n satisfies Bad, i.e., is a bad state. This means that a bad state is reachable in exactly n steps. Conversely, if formula (1) is unsatisfiable, no bad state is reachable in exactly n steps. By varying n, we can check reachability for any fixed number of steps. This shows how to reduce finite-horizon reachability to a SAT/SMT problem. Other important verification techniques that leverage SAT/SMT solvers are discussed next.

6.10 Pre-conditions, post-conditions, Hoare triples

In his seminal 1969 paper[31] *An axiomatic basis for computer programming* [83], and following earlier work by Robert Floyd [70], Tony Hoare proposed a very elegant method for proving properties of computer programs. The essence of the method consists in stating and then proving so-called *Hoare triples*. A Hoare triple is a claim of the form[32]

$$\{\phi_{\mathsf{pre}}\}\ P\ \{\phi_{\mathsf{post}}\}$$

where P is a program, and $\phi_{\mathsf{pre}}, \phi_{\mathsf{post}}$ are two formulas on the state variables of the program, called a *pre-condition* and a *post-condition*, respectively. The notation $\{\phi_{\mathsf{pre}}\}\ P\ \{\phi_{\mathsf{post}}\}$ denotes the following claim: *if program P starts at any state satisfying ϕ_{pre}, and if P terminates, then it terminates at a state satisfying ϕ_{post}.*

[31]This six-page paper is available on canvas and in your reading list: read it.

[32]Hoare's 1969 paper [83] uses the notation $\phi_{\mathsf{pre}}\ \{P\}\ \phi_{\mathsf{post}}$, but the notation $\{\phi_{\mathsf{pre}}\}\ P\ \{\phi_{\mathsf{post}}\}$ has been prevalent in recent years, e.g., see [9], volume 2, section *Hoare Logic*.

In pseudo-logic, we may write such a triple as a "formula" of the form $(\phi_{\mathsf{pre}} \wedge P_{terminates} \wedge P) \rightarrow \phi_{\mathsf{post}}$.[33,34] It is important to realize that the pre-condition is an assumption, whereas the post-condition is a guarantee.

Let us present a few examples. Suppose P is just the assignment $x := x + 1$. Then, can you tell which of the following triples are true and which are false, and why?

1) $\{x = 0\}\ P\ \{x = 1\}$
2) $\{x = 0\}\ P\ \{x > 0\}$
3) $\{x > 0\}\ P\ \{x > 0\}$
4) $\{x > 0\}\ P\ \{x > 1\}$
5) $\{\mathsf{true}\}\ P\ \{x > 0\}$
6) $\{\mathsf{false}\}\ P\ \{x > 0\}$

It is worth noting that if $\{\phi_{\mathsf{pre}}\}\ P\ \{\phi_{\mathsf{post}}\}$ holds, then: (1) $\{\phi_{\mathsf{pre}}\}\ P\ \{\phi'_{\mathsf{post}}\}$ also holds for any ϕ'_{post} which is logically weaker than ϕ_{post}, i.e., for any ϕ'_{post} such that $\phi_{\mathsf{post}} \rightarrow \phi'_{\mathsf{post}}$ (so, for example, if $\{x > 0\}\ P\ \{x > 1\}$ holds then $\{x > 0\}\ P\ \{x > 0\}$ also holds); and (2) $\{\phi'_{\mathsf{pre}}\}\ P\ \{\phi_{\mathsf{post}}\}$ also holds for any ϕ'_{pre} which is stronger than ϕ_{pre} (so, for example, if $\{x \geq 0\}\ P\ \{x > 0\}$ holds then $\{x > 0\}\ P\ \{x > 0\}$ also holds). In general, we want the strongest possible post-condition and the weakest possible pre-condition.

The beauty of the method of Hoare triples is that checking whether a triple is true or false can often be reduced to a satisfiability problem, and therefore given to a SAT/SMT solver to solve. The essence of this reduction is given by our pseudo-logical "formula" above. For instance, consider the triple

$$\{x = 0\}\ x := x + 1\ \{x = 1\}.$$

This claim is true iff the formula $(x = 0 \wedge x' = x + 1) \rightarrow x' = 1$ is *valid*, i.e., iff this formula evaluates to true for all x. By the rules of logic, a formula is valid iff its negation is unsatisfiable (UNSAT). Therefore, $(x = 0 \wedge x' = x + 1) \rightarrow x' = 1$ is valid iff $(x = 0 \wedge x' = x + 1) \wedge \neg x' = 1$ is UNSAT. Indeed, the latter formula is UNSAT: we cannot find values for x and x' to make the formula true. Therefore, the original triple is true.

Now, this was a bit too simple, because our program P was trivial in this case: just an assignment, and in fact a very simple assignment. What about more complex programs? There are essentially two approaches to handle more complex programs, following the above method. One approach is to translate a program to a formula which relates *starting* state variables (representing the value of the state at the beginning of the program) to *ending* state variables (representing the value of the state when the program ends). For some "easy" programs, e.g., say programs that have only assignments and if-then-else statements, with no loops, it is easy to see how this can be done (how?). For more realistic programs, like Solidity programs, this translation is less obvious. A key problem is how to handle loops. Luckily, many Solidity programs do not have loops, or have loops with an "obvious" number of iterations, which can be "unrolled" a known number of times (this is pretty similar to the unrolling of the transition relation in bounded model-checking, c.f. §6.9). This approach is followed by the company Certora [140].

The second approach is to propagate post-conditions backwards across the code of the program, as proposed in [83]. This process can be mostly automatized, except for loops. We will illustrate both these approaches in class.

Although the pre-/post-condition verification approach has been introduced in the 1960s, it is only relatively recently that it has become practical, thanks to the advances in SAT/SMT solvers. Verification frameworks that use this approach include:

- Certora for Solidity [140].
- SMTChecker, the Solidity compiler's formal verification engine: `https://docs.soliditylang.org/en/latest/smtchecker.html`. See also [15, 120, 121, 169].

[33] $(\phi_{\mathsf{pre}} \wedge P_{terminates} \wedge P) \rightarrow \phi_{\mathsf{post}}$ is not a real formula because it's not well-typed. In particular, P is a program, not a formula, and $P_{terminates}$ is neither.

[34] It is also worth pointing out that the above semantics of Hoare triples are the so-called *partial correctness* semantics, where the postcondition is required to hold only if the program terminates. A stronger interpretation of Hoare triples, called *total correctness*, states that if the precondition holds then the program *must* terminate (and also satisfy the postcondition). In this document, we use the partial correctness semantics.

- Verus for Rust: `https://github.com/verus-lang/verus`.
- Dafny: `https://github.com/dafny-lang/dafny`.

Solidity `assert` vs `require` Pre- and post-conditions are fundamental concepts and they are slowly making their way into mainstream programming languages. For instance, Rust has macros `requires` and `ensures` to express pre- and post-conditions, respectively – `https://doc.rust-lang.org/core/contracts/index.html`. Solidity has the `require` and `assert` statements. Both take as input a Boolean condition, and if that condition is violated, both result in a runtime exception (revert), albeit of different kind (*error* in the case of `require` vs *panic* in the case of `assert`). But their intended meaning is very different. Quoting from [10]:

> "*Assert should only be used to test for internal errors, and to check invariants.*"
>
> "*Language analysis tools can evaluate your contract to identify the conditions and function calls which will cause a Panic.*"
>
> "*[Require] to be used for errors in inputs or external components.*"

Indeed, `require` expresses a pre-condition, whereas `assert` expresses a post-condition.

6.11 Inductive invariants

Consider a transition system. An *invariant* is a state assertion that holds at all reachable states. An *inductive* invariant is an invariant which is preserved by the transition relation [110]. More precisely, an invariant Φ is inductive if, for any two states s, s', if s satisfies Φ and s' is a successor of s, then s' also satisfies Φ. What's cool is that checking *inductiveness* can be reduced to a satisfiability problem. In particular, let $(X, \mathsf{Init}, \mathsf{Next})$ be a symbolic representation of our transition system. Recall that Init is a state assertion, i.e., a formula over X; and Next is a formula over $X \cup X'$ (current-state plus next-state variables). Suppose that our candidate inductive invariant is also represented as a state assertion Φ, i.e., Φ is a formula over X. Similarly to what we did in §6.9, we use the notation $\Phi[X']$ to denote Φ but with X variables renamed to X' variables. Then, checking whether Φ is indeed inductive amounts to checking whether the following formula is valid:

$$(\Phi \wedge \mathsf{Next}) \to \Phi[X'] \tag{2}$$

Equivalently, Φ is inductive iff the formula $\Phi \wedge \mathsf{Next} \wedge \Phi[X']$ is UNSAT. And we can check this with a SAT/SMT solver!

Formula (2) only ensures that Φ is inductive: it does not ensure that Φ is an inductive invariant. For that, we need to also ensure that Φ is an invariant. As it turns out, if we know that Φ is inductive, then all we need to do is check that every initial state satisfies Φ. Why? Because if every initial state satisfies Φ, then (by inductiveness of Φ) every state that can be reached from some initial state in 1 step also satisfies Φ (make sure you understand this). But then, every state that can be reached from some initial state in 2 steps also satisfies Φ (why?). By induction on the number of steps, we can therefore show that every reachable state satisfies Φ. Therefore, Φ must be an invariant, and since we know it's inductive, it is an inductive invariant. It actually gets even better: proving that every initial state satisfies Φ can also be reduced to a SAT problem. This is equivalent to proving that the following formula is valid:

$$\mathsf{Init} \to \Phi \tag{3}$$

or equivalently, that $\mathsf{Init} \wedge \neg\Phi$ is UNSAT. We can check this too with a SAT/SMT solver!

Inductive invariants are an extremely important concept in formal verification, both theoretically and practically. To begin with, note that the set of reachable states of any transition system is an inductive invariant of that system (why?). In fact, it is the *smallest* (in set-theoretic sense) inductive invariant of that system. Second, suppose we want to prove that all reachable states satisfy a given state assertion Safe (for instance, Safe might be $\neg\mathsf{Bad}$, for given state assertion Bad). If Φ is an inductive invariant such that

$$\Phi \rightarrow \mathsf{Safe} \tag{4}$$

is valid, then all reachable states satisfy Safe (why?).

What all this means is that we can solve the reachability problem without doing explicit-state model-checking. If we can find some state predicate Φ satisfying conditions (2), (3), and (4), we are done! Plus, we can check all these conditions using SAT/SMT solvers! We will call a state predicate that satisfies all three conditions (2), (3) and (4) an *inductive invariant stronger than* Safe, or I2S2.

As you might have guessed, finding an I2S2 is not always easy. Devising methods to aid in finding inductive invariants, including computer-aided inductive invariant inference, is an active area of research, and an area some of my students have worked on [146, 147, 144, 145].

We end this section by mentioning an important model-checking algorithm, IC3 [35]. Roughly, IC3 works by strengthening Safe until it becomes inductive, while also over-approximating the set of reachable states. Therefore, IC3 can be seen as computing automatically a safe inductive invariant. (Note that the same is true for an algorithm that computes the set of reachable states, as this set is always the strongest inductive invariant; however, the set of reachable states may be too difficult to compute, which makes computing an over-approximation of it interesting.) State of the art inductive invariant inference techniques are directly inspired by IC3 [147].

6.12 Safety and Liveness

So far in this document, and for the most part in this course, we talk about the simple verification problem of checking state assertions. State assertions are very useful properties to check; arguably, much of formal verification consists in checking state assertions. But state assertions alone are not enough. There are many more properties that we would like our systems to have.

6.12.1 Safety

To begin with, we would like to verify general *safety* properties. Intuitively, a safety property is a property that states that nothing "bad" ever happens. State assertions are safety properties. They state that a "bad" state (i.e., a state that violates the assertion) is never reached. But there are other safety properties that cannot be immediately expressed as state assertions. For example, consider an ERC-20 token contract (§7.4). A safety property for this contract might be the following: *the sum of all token* `balances` *is not changed by* `transfer`. This is a safety property because it states that a bad thing never happens: the bad thing would be that `transfer` changes the sum of all `balances`. (We assume that `balances` is a mapping that holds the token balance of each account.)

How to check this property? We need to know what the sum of all `balances` is right before `transfer` is called, but also right after `transfer` finishes; then we need to check that the two sums are equal. Suppose we already have a state variable `totalBankBalance` that gives us the (current) sum of all `balances`. This state variable alone is not enough: we need to *memorize* the value of `totalBankBalance` right before `transfer`, and check that the value of `totalBankBalance` right after `transfer` is equal to the memorized value. Therefore, we need an extra state variable to store the memorized value.

As this discussion shows, safety properties generally require extra state variables. This is in fact both necessary and sufficient. With enough additional state variables, we can reduce all safety properties to state assertions. Here's another example of a safety property: *every request is granted within time* T. To reduce this property to a state assertion, we need to set a countdown timer to T when a request occurs. Then we need to decrement the timer every time unit (every second, or every minute, etc, depending on what accuracy we require). If the timer reaches zero before the request is granted, we have a safety violation. Then, the state assertion could be something like $\texttt{no\_request\_pending} \vee \texttt{timer} > 0$ where `no_request_pending` is set to false when a request arrives and to true when it's granted.

These additional state variables, plus the extra transitions required to update them, can be seen as state machines that *monitor* the system that we want to verify. They are called *safety monitors*. Verification tools like Certora where the system under verification is a real program (e.g., in Solidity) have means to add these monitors. For instance, in Certora the monitor state variables are called *ghost variables* and the transitions to update ghost variables are called *hooks* – `https://docs.certora.com/projects/tutorials/en/latest/lesson4_invariants/ghosts/basics.html`.

6.12.2 Liveness

Safety is not enough. This holds for systems, as well as in politics and life in general. Safety says that something bad will never happens. It is easy for a system to satisfy this by doing nothing. A car that refuses to leave its garage is a very safe car because it avoids collisions. An airport that grounds all planes is a very safe airport. Clearly, these systems are useless, which means we need something more than safety.

Liveness answers this need. A liveness property states that something "good" will happen. For example, the car will eventually start to move and will take us where we want to go. Just like safety, liveness alone is not enough: we want the car to move (liveness) but we also want it to avoid collisions (safety).

In English, many liveness properties are expressed using the word *eventually*. For example, *every request is eventually granted* is a liveness property, called a *request-response* property. Request-response is liveness, and not safety, because we don't necessarily know *by when* the request will be granted, and therefore we don't require a specific time limit. It is interesting that when we do require a time limit, as we did in §6.12.1, request-response becomes a safety property! Program termination is a typical liveness property of the request-response type. We may not care or not know how long it will take for the program to execute. But we still want the program to terminate once it starts.

Our discussion of safety vs liveness is informal. See [14] for an elegant formal characterization.[35]

6.12.3 Fairness

In order to satisfy liveness, we typically need to make some assumptions. These are called *fairness* assumptions. For example, suppose we model a communication protocol like TCP, which retransmits messages whenever there is a timeout. We want to ensure that this system satisfies the liveness property *every message is eventually delivered*. Because messages can be lost, this property does not hold in general. There are behaviors, for instance, where a message is lost, our protocol retransmits it, the retransmission is also lost, our protocol retransmits again, the second retransmission is also lost, and so on, forever. So the message never gets delivered. In reality, of course, the probability of this happening is zero, because real networks do not keep losing messages forever. But our model is non-deterministic and does not have probabilities. To overcome this, we need to make a fairness assumption, namely, that messages cannot be lost forever. Fairness properties are liveness properties. The property *messages cannot be lost forever* could be re-stated as *eventually, some message will not be lost*.

6.13 Temporal logic

We talked about safety and liveness, but we did not talk about how to verify liveness properties. Before we can begin to address this question, we need to discuss how to formally specify liveness properties in the first place. An elegant way to specify liveness (as well as safety and other) properties is using *temporal logics*. These are logics developed specifically in order to formally define properties of dynamical systems like state-transition systems, that is, properties of systems whose state changes over time.

There are many temporal logics. Some of the most well-known are *linear temporal logic* (LTL) [129] and branching-time logics [65]. For example, the property *every r is followed by g*, for state assertions r, g, is written in LTL as $\mathbf{G}(r \rightarrow \mathbf{F}g)$. The property *infinitely often r* is written in LTL as $\mathbf{GF}r$. The property

[35]Among other things, [14] shows that every property can be expressed as the intersection of a safety property and a liveness property.

almost never g may be written in LTL as $\mathbf{FG}\neg g$. All these are liveness properties. The property $\mathbf{G}(x > 0)$ for state variable x is the safety property *x is always greater than zero.*

The subject of temporal logic is fascinating, but we will only touch upon it in this class, due to time constraints. The same is true for verification of such logics, and verification of liveness properties. See [24, 47, 109, 110, 111] for more in-depth discussions of these topics.

6.14 Logic vs natural language

Natural language (English, French, Greek, etc) is great for songs and literature, but it lays ambiguity and misunderstanding traps that can be disastrous, in love, politics, and engineering. You will have the opportunity to see some of the pitfalls of natural language in one of your assignments. Temporal logic offers more examples of such pitfalls. For instance, consider again the property *every request is eventually granted.* Assuming that r and g are state assertions denoting the arrival and granting of a request, respectively, we might write this property in LTL as $\mathbf{G}(r \to \mathbf{F}g)$. But this formula is satisfied even if the number of g's is not equal to the number of r's! Indeed, according to the semantics of LTL, a g "cancels" all r's preceding it (in fact, it also cancels an r happening at the same step as the g). Does this formula express what we want? The answer to this is *it depends.* It depends on what the English phrase *every request is eventually granted* actually *means.* This is an example of natural language ambiguity.

Now consider the phrase *every r is followed by g.* This seems to correspond more closely to $\mathbf{G}(r \to \mathbf{F}g)$. But still we have another ambiguity: what if r and g happen at the same time? Does this count as g having "followed" r or not? Again, the English phrase is ambiguous. If an "instantaneous" grant counts, then $\mathbf{G}(r \to \mathbf{F}g)$ works; if not, then we need to use LTL's *next* operator, and the formula $\mathbf{G}(r \to \mathbf{XF}g)$ which can be phrased as *every r is followed by g but starting from the next step.* Yet a third interpretation is to interpret "followed by" to mean not *starting from* the next step, but *precisely at* the next step. In that case, we need to use the LTL formula $\mathbf{G}(r \to \mathbf{X}g)$. This example again illustrates the ambiguity of natural language and how formal logic can help resolve ambiguities. There are many more examples. For instance, paper [87] (by ♠ Holzmann, creator of Spin) discusses the possible formalizations of the informal requirement "*when you pick up the phone, you get a dialtone*". Automated translation of natural language into logic, and in particular of informal requirements into formal specifications, is an active area of research, e.g., see [40, 82, 19].

6.15 Synthesis

Synthesis goes beyond verification and aims to generate correct-by-construction software [45, 130, 134, 68, 78, 64]. This section will be populated later. FIXME

6.16 Other verification methods and topics

There are many different formal verification methods. In this course, we got a glimpse of theorem-proving (§6.6), and discussed in a bit more depth model-checking (§6.5), pre-/post-conditions and Hoare triples (§6.10), and inductive invariants (§6.11).

But the field is vast, and there are many more formal verification methods out there; such as (the following ♠ citations are just representative samples; there's a large body of work in each of these topics):

- Equivalence checking of digital circuits [113]
- Simulations, bisimulations, ... [112, 72]
- Refinement [22]
- Abstract interpretation [50]
- Compositional verification [54]
- Probabilistic verification [23, 25]

- Timed and hybrid systems [17, 16]
- Formal logics and verification methods for security and cryptography[36]

and many more [24, 47], which we will likely not have time to talk about in this class. See §6.2 for other courses that I teach on this topic. See also the lecture notes of my CS-2800 course [163]. See `https://slebok.github.io/proverb/index.html` for a large list of tools related to verification.

In §6.2 we contrasted testing to proofs. Although we do not discuss testing in this course, testing is a key tool in developing software. Therefore we consider formal verification as complementary, and not an alternative to testing. Testing is very good at finding bugs, which is extremely useful, whether or not we care about proving correctness. (Note that some experience reports claim that formal verification is also good at finding bugs [116, 140].) Also, testing methods have improved considerably over the years (e.g., see [30, 31] and references therein). Testing techniques like *fuzzing* are widely popular [108]. And when formal specifications are available, **conformance testing** or **property-based testing** techniques can be used to automatically generate tests from these specifications [36, 93, 76]. Availability of formal specifications also enables **runtime verification** (RV) [81]. RV consists in compiling a formal specification into a *monitor* which observes the real system at runtime, and alerts whenever the specification is violated. This is in contrast to formal verification which checks correctness of a model of the system instead of the real system.

Currently there are many efforts to leverage AI in formal methods and mathematics, e.g. see [158], `https://foundation.tlapl.us/challenge/index.html`, `https://icml.cc/virtual/2025/workshop/39979`, `https://www.nsf.gov/funding/opportunities/aiming-artificial-intelligence-formal-methods-mathematical`, `https://www.renaissancephilanthropy.org/initiatives/ai-for-math-fund`, `https://www.galois.com/articles/claude-can-sometimes-prove-it`.

7 Decentralized Finance – DeFi

The birth of cryptocurrencies can be considered to have happened in 2008 with the introduction of Bitcoin [115].[37] 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. ♠

7.1 What is money?

revise (scarcity, conservation, etc) 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

[36] There are several security/cryptography verification tools that we won't discuss in this course:

– *Tamarin*: `https://tamarin-prover.com/`.

– *ProVerif*: `https://bblanche.gitlabpages.inria.fr/proverif/`.

– *EasyCrypt*: `https://www.easycrypt.info/`.

– *SSProve: A Foundational Framework for Modular Cryptographic Proofs in Coq*: `https://eprint.iacr.org/2021/397`.

– *zkLean* [123].

Also see [48, 12, 13, 44, 133] and references therein.

[37] 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" [51].

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 [20, 26, 118], 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 fuel 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.

7.2 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"[20]; "Money [...] has a value that people trust"[26]; "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."[118] [38]

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 ...*). When trust erodes the entire system is in danger of collapsing, as many of us who have lived through the recent financial crises know.

7.3 The double-spending problem

In addition to classic trust issues like the above, digital money and cryptocurrencies raise new challenges not found in classic money. One of these challenges is beautifully illustrated by the *double-spending problem* – c.f. [115], section *Transactions*.

[38] 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); and second, commercial banks also create money.

Central banks create ("print") money, e.g., by giving loans to their account holders. For example, if some entity A has an account with the Federal Reserve (A might be a large commercial bank or a government agency), and the Federal Reserve issues a loan of 1 million USD to A, then 1 million USD is added to A's account: effectively, 1 million USD fresh money has been created, and added to circulation since A can now start using this money. But central banks are not the only entities that are allowed to create money: commercial banks also create money by issuing loans. According to [26]: "Commercial bank money makes up most of the money that people actually use. It is backed and balanced by the assets held by commercial banks. So it is created by banks expanding their balance sheets. For example, if a commercial bank grants you a loan to buy a car, they create money in this way. And when you repay the loan, the money created disappears."

You might think that when a commercial bank gives a loan it does not create new money, because it in fact loans the money that it holds in deposits. For example, A might have a savings account with some bank. A has \$10,000 in their savings account. Then B requests a loan of \$5,000 from the same bank, and the bank grants B the loan. Now, B's account has \$5,000 and A's account still has \$10,000. So in total there is now \$15,000, compared to \$10,000 prior to the loan, i.e., \$5,000 new money has been created. But, you may wonder, does the bank actually still have all of A's money? What if A decides to withdraw all their money? These are very legitimate questions and they lead to questions such as *how much of the depositors' money should banks be allowed to lend?* These are fascinating topics in economics and monetary policy which unfortunately are beyond the scope of this class. Wikipedia and other online sources have many informative articles on the topic, e.g., https://en.wikipedia.org/wiki/Money_creation, https://en.wikipedia.org/wiki/Reserve_requirement, https://en.wikipedia.org/wiki/Gold_standard, https://www.investopedia.com/terms/l/loan-to-deposit-ratio.asp.

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, Alice can try to reuse (*double-spend*) the (same) coin. Indeed, Alice can go to Chris, get one of his delicious cookies, and transfer the same 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 problem 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 reuse it. The problem does not arise with digital means of payment that go through a centralized point, either. Consider, for instance, your credit card. When you pay for something, you don't give away your physical credit card, nor the digital card on your phone. You tap your card, and keep it for the next transaction. But you cannot double-spend, 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 use your card again to buy another $100 worth of something, your second transaction will fail. It 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 second request.

But note that we want to have digital money *without* a centralized authority. How can we then solve the double-spending problem? You will be asked to examine this question as part of your assignments.

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 – §8.

7.4 Ethereum Tokens: money standards

As discussed in §7.1, money is something that can be exchanged. But how exactly are these exchanges made and what rules must they obey? Ethereum *tokens* provide an elegant, canonical, *computer-sciency* answer to this question: *money is a smart contract that implements a token standard*.

A *token* is any kind of asset that can be traded (exchanged) on Ethereum – https://ethereum.org/eth/. There are many kinds of tokens, including *fungible* tokens and *non-fungible* tokens (NFTs). Two units of a fungible token are indistinguishable from each other: one ether is equivalent to any other ether, just like a physical dollar is equivalent to any other physical dollar. NFTs are "individually unique" – https://ethereum.org/nft/.

Our focus here is on fungible tokens. These are governed by token standards – https://ethereum.org/developers/docs/standards/tokens/. An important standard is ERC-20 – https://ethereum.org/developers/docs/standards/tokens/erc-20/ and [168]. In a nutshell, ERC-20 defines a Solidity *interface*. An ERC-20 Token is a contract that implements the ERC-20 interface. We use the term *interface* here in the object-oriented sense. A Solidity interface is like a Java interface: it is a set of *abstract* functions (function signatures without implementation, i.e., without function bodies). An implementation of the interface provides the missing body to each of the functions of the interface. Example implementations of ERC-20 and other token standards are available online, e.g., by OpenZeppelin – https://docs.openzeppelin.com/contracts/5.x/api/token/erc20 – and others. A list of contracts implementing ERC-20 is provided by Etherscan: https://etherscan.io/tokens.

It is important to understand the difference between cryptocurrencies and tokens. For example, ETH (ether) is not the same as WETH (wrapped ether). ETH is the currency of Ethereum, whereas WETH is an ERC-20 token: see https://ethereum.org/wrapped-eth/.

An interesting question is what exactly is an Ethereum "standard" and how does the Ethereum standardization process work. In short, this is done via EIPs (Ethereum Improvement Proposals) – https://ethereum.org/developers/docs/standards/. EIPs are reminiscent of the RFCs (Requests For Comments) of the Internet – https://www.ietf.org/process/rfcs/. Ethereum standardization is related to questions like *who controls/owns/governs Ethereum?* We may touch upon these questions in §**??**. For now, see https://ethereum.org/governance/.

7.5 DAOs, Wallets, Flash loans, Leverage, Liquidity pools, AMMs, DEXs, Stablecoins, Bridges, Auctions, Options, ...

7.5.1 Leverage

We explain leverage only through an example. Suppose I have \$10 to start with. I strongly believe that the price of StavrosCoin (STC) will go up, so I spend my \$10 and buy 10 STC. Now I have 10 STC. Currently, 1 STC = \$1, so if I sell back my 10 STC, I'll simply get back my \$10 (minus a fee) so nothing really happens (except that I lost money = the fee). But I don't sell my STC. Instead, I use these 10 STC as collateral, to borrow, say, \$9. What this means is that if I'm not able to pay back the \$9 (plus perhaps a fee) by a certain deadline, I will lose my 10 STC. Let's call this *Loan 1*. So Loan 1 = \$9. Now I have \$9 that I just borrowed. Since I believe so much in STC, I spend these \$9 to buy 9 fresh STC. In total I now have 19 STC. (But I also owe \$9 from Loan 1.)

I use my 9 fresh STC to get a new loan, Loan 2, where I put the 9 STC as collateral and borrow \$8. (I cannot put the entire 19 STC as collateral, because 10 of those are collateral for Loan 1, and the lender won't let me use the same collateral twice.) So, Loan 2 = \$8.

I continue in the same way, using the \$8 to buy 8 more STC, using the 8 STC as collateral for Loan 3 = \$7, using the \$7 to buy 7 more STC, using them as collateral for Loan 4 = \$6, and so on. The process finishes when I use my last 2 STC to borrow Loan 9 = \$1, and this \$1 to buy 1 more STC (which is not enough to use to borrow any more \$s, because nobody will lend me any \$s for just 1 STC). When this process finishes, I have:

- $10 + 9 + 8 + 7 + \cdots + 1 = 55$ STC.
- Loan 1 + Loan 2 + ... + Loan 9 = \$ 9 + 8 + 7 + 6 + 5 + 4 + 3 + 2 + 1 = \$45.

Now, suppose that STC doubles in price, so now 1 STC = \$2. Then, I can sell $22 + 6 = 28$ STC, and get back $2 \cdot 28 =$\$56. With these \$56 I can pay my entire loan debt of \$45, and still have \$11 left (a bit more than my original \$ amount). Plus I also have $55 - 28 = 27$ STC. So I have a net profit of 27 STC. If I were to convert that to \$, it would be $2 \cdot 27 =$\$54. That's a net profit of 5.4 times my original amount of \$10, i.e., a net profit of 540%. Note that this is much bigger than if I had just bought 10 STC with my original \$10, and then sold my 10 STC to get back \$20. Then I would have made only \$10 more, i.e., a net profit of only 100%, compared to 540%. That's what leverage "buys" me.

On the other hand, suppose the STC falls in price. Typically, my lenders will require me to pay back my loans as soon as STC falls even by as little as say 10%. (This will be a clause in the loan.) Since I have \$0 left, I will have to default on all my loans, which means I will lose all my STC except the last 1 STC which I couldn't borrow money with. So in this case, I lost all my \$s and I am left only with 1 STC, which is worth now less than \$1. So I lost more than 90% of my original \$10. If I had simply bought 10 STC without leverage, I could now sell them back and get back a bit less than \$9, say, \$8.5. In that case, my losses would be only 15% instead of 90%.

7.6 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 [61]. Notably, although Diem is dead, Move seems to still be alive – see https://github.com/move-language/ ♠

`move`.

8 Blockchains

We have talked about Solidity smart contracts in §4, and how these contracts use Ethereum concepts such as addresses, accounts, ETH balances, etc – §4.2. But what exactly is a blockchain and how does it accomplish the promise of decentralized trust (c.f. §1.2)?

There is actually many ways to answer what a blockchain is, depending on one's viewpoint. A Solidity programmer may look at the Ethereum blockchain as a centralized state machine – §8.1. This abstract view hides implementation details and allows the programmer to focus on *what* Ethereum provides and not *how* it achieves it. On the other hand, someone who wants to participate in the Ethereum network, say by running their own Ethereum node, may want to understand exactly how the Ethereum protocols run. In what follows we examine several of these different perspectives and how they relate to each other.

8.1 The centralized state machine view

From the point of view of a smart contract programmer (or that of a holder of an external Ethereum account) a blockchain like Ethereum is a single, centralized, *state machine* – c.f. §6.4.1. Specifically, the EVM, whereupon Solidity programs run, is just one big state machine. It is big because its state is very large: its state contains the bytecode of every single contract deployed on Ethereum; it also contains all the (storage) state variables of all contracts; it also contains the ETH account balances of all Ethereum accounts.

The transition function of this state machine is also very large: every possible Ethereum *transaction* corresponds to a transition (change of state) of the machine. Every possible transfer of ETH from one account to another is a possible transaction (and transition of the machine). Every possible invocation of any public function of a contract from an external account is also a possible transaction/transition. (Note that the invocation of a contract's function may itself trigger other function calls within the same contract or in other contracts. The transaction will then consist of the entire sequence of calls.) The creation (deployment) of a new contract is also a transaction/transition.

The following remarks about transactions are in order (c.f. `https://ethereum.org/developers/docs/transactions/`):

- Ethereum transactions can only originate from EOAs. Quoting [170]: "The sender of a transaction cannot be a contract."
- Transactions cost gas: `https://ethereum.org/developers/docs/gas/`. Among other things, gas prevents certain denial-of-service attacks; for instance, where an attacker tries to overwhelm the system by flooding it with transaction requests, or by submitting transactions that take too long to execute.
- Each transaction is cryptographically *signed* by the sender of the transaction (e.g., using the sender's private key, see §9.2). Ethereum checks the signature (e.g., using the sender's public key) prior to selecting a transaction. This avoids attacks where Eve tries to impersonate Alice, e.g., by sending a transaction that says *Alice sends 1000 ETH to Eve.*

A precise formalization of EVM/Ethereum's global state (or *world state* in [170]) as well as its transition function is given in the so-called Ethereum *yellow paper* [170]. (The Ethereum yellow paper [170] is different from the Ethereum white paper [38].) That formalization describes in detail what information exactly is included in the global state and also in a transaction, and how exactly each transation modifies the global state. It also specifies the rules that make a transaction *valid*. For instance, a function call that reverts will result in an invalid transaction; a transaction that runs out of gas is also invalid; a transaction that attempts to transfer more ether than is available in an account is also invalid; and so on. Honest nodes, that implement the EVM/Ethereum rules correctly, ensure that only valid transactions get to be included in the blockchain (more on this later).

The centralized state machine perspective is an illusion, but a very useful illusion. It is an illusion because in reality the blockchain state machine is implemented in a decentralized manner – c.f. §8.3. At the same time, thinking about this state machine as if it were centralized is very useful. It is a programming abstraction which helps smart contract developers reason about their code without having to worry about how EVM/Ethereum are implemented, or how they work "under the hood." In fact, the beauty of blockchain protocols is that they hide the decentralization aspects, and present to the user as much as possible a centralized view.

8.2 The centralized blockchain data structure view: a permanent, public, append-only ledger

At this point you may be wondering: *OK, the state machine view makes sense, but it talks neither about blocks nor about a chain. Isn't a blockchain a chain of blocks? Where do those come in?*

Indeed, a blockchain is a chain of blocks. For now, we can view it as a single, centralized data structure consisting of a unique sequence of blocks. This again is a useful abstraction. The reality is more complicated: the blockchain is a decentralized data structure maintained by the nodes of a peer-to-peer network, but we may ignore this for now. The blockchain starts at B_0 (the *genesis* block), and continues with $B_1, B_2, ...$, up to some current block B_n. Importantly, the chain is *append-only*: we can add a new block B_{n+1} after the current block, but that's the only modify operation we can do to the data structure. We cannot delete blocks, and we cannot modify the contents of existing blocks either. The blockchain is *public* in the sense that anyone can read it. What about writing? Anyone can *submit* transactions to be included to the blockchain. But in order for a transaction to be included, certain things need to happen (§8.2.5).

A block B_i can be viewed as an ordered sequence of transactions $t^i_1, t^i_2, ..., t^i_{k_i}$. Because transactions within a block are strictly ordered, the entire blockchain can be viewed as a long sequence of transactions: $t_1, t_2, t_3, ..., t_N$. A transaction can transfer ETH from one account to another, create (*deploy*) a contract, or call a function of a contract. Each of these operations changes the global state of the EVM/Ethereum, that is, it corresponds to a transition in the state machine view described in §8.1. Therefore, if we know the initial state s_0 of the EVM/Ethereum, we can compute the next state s_1 after t_1, then s_2 after t_2, and so on up to t_N, which will give us the current state s_N of the EVM/Ethereum. We can now see how this blockchain view reconciles with the state machine view: the blockchain records the history of blocks/transactions, i.e., the history of transitions of the state machine; and in a deterministic state machine, knowing the initial state and the history of transitions suffices to compute the current state – §6.4.1.

This view omits several important aspects of blockchains. The most important aspect is decentralization, which is what makes blockchains interesting in the first place. Decentralization is discussed in §8.3 and subsequent sections. Some other aspects are discussed next. As with the rest of these lecture notes, our goal is not to be exhaustive, but to bring up the most salient concepts.

8.2.1 Why blocks instead of just transactions?

Why do we need blocks? Why not publish transactions on the blockchain directly? For efficiency reasons: appending something to a blockchain is a non-trivial operation because of decentralization (as we shall see later). It is therefore more cost effective to group several transactions together in a block and append them all at the same time.

8.2.2 Where exactly is the EVM/Ethereum global state stored?

Does each block store the entire state of the EVM/Ethereum state machine? No. Ethereum blocks do not store the state of the EVM/Ethereum state machine. As discussed in §8.1 this state is huge. Therefore, storing it in the blocks would make the blocks also huge, and the blockchain even huger. This would be prohibitive in terms of memory, so blocks do not store the state: they only store the transitions = the transactions. You can think of blocks as only storing the "diffs".

But this raises another issue. If the global state is not stored anywhere, does it have to be recomputed from scratch every time? Do we need to start with s_0 and apply all the transactions $t_1, ..., t_N$ everytime we need to know something about the current state s_N? That would be prohibitive in terms of time. Instead, some Ethereum *nodes* (but not blocks!) indeed store the entire EVM/Ethereum state.[39] These *nodes* are the computers (hardware plus software) that make up the Ethereum network. There are many kinds of nodes.[40] Nodes that maintain the global state update that state incrementally, one transaction at a time.

8.2.3 Do we actually need the EVM/Ethereum global state?

Yes, for many things. For instance, each time you query the ETH balance of some account, e.g. with `cast balance ...`, you are reading the global state. You are *not* reading a block, since the block does not hold the account balances, it only holds transactions. If you wanted to figure out the account balance based solely on the blockchain you would have to traverse the entire blockchain, starting from the origin block all the way to the current block, find all transactions that this account has been involved in, and derive from them all credits and debits to the account. Clearly, this is unmanageable. On the other hand, if you know the global state, returning the balance of some account is essentially a lookup operation in some table.

The global state is needed, crucially, to validate proposed transactions. If Bob has only 1 ETH in his account and tries to submit a transaction saying *Bob sends 2 ETH to Alice*, this transaction will not go through. The EVM will attempt to process the transaction starting from the current global state, and will find out that Bob does not have enough ETH to send to Alice. As we can see, the EVM needs to know the current state, in order to validate transactions (and compute the next state for valid transactions).

8.2.4 Why is the global EVM/Ethereum state stored as a *modified Merkle Patricia tree*?

The global EVM/Ethereum state is stored as a *modified Merkle Patricia tree* [170]. But what is this tree and why do we need it? This tree is a data structure where each node of the tree has the hash of its children. So the root of the tree has the hash of the entire contents of the tree. This allows to compare two trees in constant time, independently of the size of the trees: instead of comparing the entire contents of the two trees, we only compare their root hashes. If the two root hashes are different, the trees must be different. If the two root hashes are equal, the trees are equal with a high probability (by the properties of hash functions, there is a small chance that two different inputs hash to the same output, so it could be that two different trees have the same root hash, but this is very unlikely). This data structure is also *tamper-resistant* in the sense that any attempt to modify any part of the tree will (very likely) result in a different root hash, revealing plainly that the tree has been modified.

The root hashes of the global EVM/Ethereum state are stored in the blocks of the blockchain. This allows to compare global states (which are huge) more efficiently. For a detailed description of how the world state ♠ of Ethereum is encoded see https://ethereum.org/developers/docs/data-structures-and-encoding/patricia-merkle-trie/ and https://blog.ethereum.org/2015/11/15/merkling-in-ethereum.

8.2.5 How are new blocks added to the blockchain?

This is one of the most fascinating aspects of decentralized blockchains, which we discuss later (§8.3 onwards). From the centralized viewpoint, the simplified answer is: (1) external users submit their (signed) transactions to the blockchain; (2) the blockchain selects some of the submitted transactions, orders them, executes them (in that order), and discards any invalid transactions (e.g., reverts); (3) the selected and valid transactions are grouped in a block (in the same order as above), and the block is appended to the end of the blockchain.

In reality, these steps are not performed by a single (centralized) entity. That would require us to trust that entity, which is what we want to avoid in the first place! We refine this simplified view from §8.3 onwards.

[39] Quoting from [170]: "The world state (state), is a mapping between addresses (160-bit identifiers) and account states [...]. Though not stored on the blockchain, it is assumed that the implementation will maintain this mapping in a modified Merkle Patricia tree." For a first discussion on why a Merkle Patricia tree is used, see §8.2.4.

[40] https://ethereum.org/developers/docs/nodes-and-clients/

8.2.6 The centralized blockchain view on Etherscan

In summary, a blockchain can be seen as a centralized, permanent, public, append-only ledger. This view is displayed on Etherscan [3]. Visit `https://etherscan.io/blocks`, browse through the blocks shown there, and map what you see to the concepts you learned above.

8.3 Decentralization

Things would be easy if a blockchain like Ethereum were maintained by a single entity. We don't want that, for a number of reasons. A single entity runs the risk of *single point-of-failure*: if the (unique) computer running Ethereum breaks down, then service becomes unavailable, data may be forever lost, etc. A single processing entity also has limited bandwidth, thus cannot scale. And importantly in the DeFi setting, if we had a single entity then we would have to trust that entity, which we don't want to do.

Therefore, we must choose a *decentralized* architecture, where the blockchain must be somehow *distributed* over many nodes.[41] What this means, for blockchains like Bitcoin or Ethereum, is that the blockchain is *replicated* over many nodes. That is, there are many computers, each of which maintains their own copy of the blockchain (in the case of Ethereum, each computer also runs their own copy of the EVM). But if there are many different blockchain copies, the centralized blockchain data structure abstraction (§8.2) breaks down. And if there are many EVM copies, the centralized state machine abstraction (§8.1) breaks down, too. In order to maintain these abstractions, we must somehow ensure that all these copies *agree* with each other. This is what *consensus* protocols achieve.

But before talking about consensus, we need to make the following point clear:

8.3.1 Concurrency and Asynchrony

It is important to understand that even if every node is honest and there are no faults, consensus is still needed in a distributed system. This is because of the *concurrent* and *asynchronous* nature of distributed systems [95]. Concurrency means that several things are happening at the same time. In a distributed system, there are processes running in parallel on separate computers, and each of these processes is doing its own thing, sending and receiving messages, changing its local state, etc. Asynchrony means that these processes are not a-priori synchronized: each process runs at its own pace, with its own local notion of time. There is no *global clock* to which all processes can refer to. Indeed, establishing a global time frame in a distributed system requires itself some form of agreement! (This is in contrast, for instance, to *synchronous digital circuits* where a single clock is distributed to all components of the circuit, thereby establishing a common notion of *clock cycle*.)

Consider, for example, a distributed blockchain with two nodes, A and B. Both are honest and there are no faults. Suppose some user U_1 submits transaction t_1, and some user U_2 submits transaction t_2. Both t_1 and t_2 are broadcast to the entire network. But it may happen that node A receives t_1 first and then t_2, whereas B receives first t_2 and then t_1. Suppose t_1 and t_2 are in conflict, for instance, t_1 requests to transfer 100 tokens from an account that has 150 tokens in total, and t_2 requests to transfer 80 tokens from the same account. Then the order of their execution matters: if t_1 is executed first, then t_2 will revert, and if t_2 is executed first, then t_1 will revert. In the end, only one of the two should be executed, but which one? It would be naive to answer *whichever is seen first* because then A would execute only t_1 and B would execute only t_2. This would result in A and B producing a different global state.

Instead, A and B need to agree on the global state, so they need to agree on the order in which transactions should be executed. For that, A and B need some consensus mechanism. As this example illustrates, consensus is needed in a distributed system, even in the absence of failures and even when there are no malicious nodes.

[41] For the purposes of our discussion, we consider the terms *decentralized* and *distributed* synonymous.

8.3.2 Crash faults, Byzantine faults, and Sibyl attacks

In addition to concurrency and asynchrony, there are other things to worry about when we have a distributed system. One such issue is how to cope with node failures (also called *faults*). There are different types of faults. Relatively easy to deal with are *crash faults*, where a node suddenly crashes and does nothing after that (in particular, it sends no more messages). Harder to deal with are *Byzantine faults* [125, 102], where a faulty node can behave pretty much arbitrarily. For instance, a Byzantine node may send no messages at all, and thus, crash faults are a special case of Byzantine faults. But Byzantine faults are strictly harder because they also allow a faulty node to send inconsistent messages to the other nodes. For example, a Byzantine node may send one message to node X and a different message to node Y, or no message to Y at all. Byzantine faults can model some network failures, e.g., not sending a message to Y is the same as having a communication failure with Y. Byzantine faults can also model some types of malicious behavior, e.g., where a node intentionally "lies" to other nodes.

Byzantine faults cannot model all types of faulty or malicious behavior. Of particular interest in the context of blockchains are *Sibyl attacks*, where a single (malicious or faulty) entity can present multiple identities to the system, thus appearing as multiple different entities [62]. For example, what happens if a malicious actor controls many of the Ethereum nodes? Sibyl-resistance mechanisms such as proof-of-work (§8.4) and proof-of-stake (§8.5) are designed to protect against Sibyl attacks.

8.3.3 Permissioned vs permissionless

In some distributed systems the set of participating nodes is known (and sometimes fixed in advance). For example, think of an airplane with say 3 computers, for redundancy. Or imagine a bank which runs its accounting system on a distributed system of 10 servers, all locked up in the basement of the bank. In this case, it is reasonable to assume that the bank has total control over the set of nodes; importantly, the bank can number the nodes, say $0, 1, ..., 9$, and implement a distributed protocol where each node knows of the other 9 nodes. This is not possible in a system like a decentralized blockchain, where any node can participate in the network, with nodes dynamically joining or leaving the network as they wish. In the literature of distributed systems, the case where all participating nodes is known is called *permissioned*, and the case where arbitrary nodes can join/leave is called *permissionless*. Both Bitcoin and Ethereum are permissionless.

8.4 Longest-chain Consensus – Proof of work

Longest-chain consensus with proof-of-work (PoW) is the way Bitcoin solves the problem of distributed agreement.[42] Quoting from the abstract of the Bitcoin white paper [115]:

> "The network timestamps transactions by hashing them into an ongoing chain of hash-based proof-of-work, forming a record that cannot be changed without redoing the proof-of-work. The longest chain not only serves as proof of the sequence of events witnessed, but proof that it came from the largest pool of CPU power. As long as a majority of CPU power is controlled by nodes that are not cooperating to attack the network, they'll generate the longest chain and outpace attackers."

The Bitcoin solution actually combines several ingenious ideas, each answering part of the puzzle, together in one protocol. We examine these ideas one by one.

8.4.1 The chain of hashes

Each block in the blockchain *cryptographically references* its parent (i.e., the previous block in the chain). What this means in a nutshell is: every block is a tuple (d, n, p, t) where d is all the data of the block (all

[42]Ethereum initially used a PoW consensus mechanism, similar to Bitcoin. In 2022, Ethereum transitioned to a proof-of-stake protocol: see §8.5.

transactions stored in the block, etc), n is a *nonce* (this is used in the proof-of-work mechanism explained below), p is the hash of the parent block, and t is the hash of this block; t is obtained by cryptographically hashing the remaining 3 elements, i.e., $t = H(d, n, p)$, where H is some cryptographic hash function – c.f. §9.1.

So, if $B_0, B_1, B_2, ...$ are the blocks in the blockchain, and supposing that each $B_i = (d_i, n_i, p_i, t_i)$, then we must have: (1) for all i, $t_i = H(d_i, n_i, p_i)$, and (2) for all $i > 0$, $p_i = t_{i-1}$.

This "chaining with hashes" results in the following crucial property: if some block B_i is modified *in any way*, then all subsequent blocks have to be modified also, in order for conditions (1) and (2) to continue to hold. For example, suppose that the data field d_3 of block B_3 is modified, say, replaced by d_3'. Then, by the properties of cryptographic hash functions, $t_3 \neq H(d_3', n_3, p_3)$, i.e., condition (1) is violated. So a new t_3 needs to be computed to fix this, say $t_3' = H(d_3', n_3, p_3)$. But now $p_4 \neq t_3'$, so condition (2) is violated. To fix this, p_4 needs to be replaced by $p_4' = t_3'$. But then $t_4 \neq H(d_4, n_4, p_4')$, which means t_4 would have to be updated, and so on. As can be seen, the updates have to be propagated all the way to the end of the blockchain, in order for the blockchain to be valid again, i.e., to again satisfy (1) and (2). In combination with the proof-of-work requirement explained next, such a propagation would be computationally too expensive.

A beautiful illustration of how the chain of hashes works is given in the interacive demo `https://andersbrownworth.com/blockchain/blockchain`. The demo itself is explained in the video available at `https://ethereum.org/developers/docs/intro-to-ethereum/`.

8.4.2 Proof of work – Miners

Cryptographic hashes are typically long strings of bits. For example SHA-256 hashes are 256-bit long strings. The proof-of-work mechanism places a requirement on the hash t of a block (d, n, p, t): *t must begin with a certain number of* 0*s*. Given that d is fixed (the transactions of the block) and p is also fixed (the hash of the parent block), to achieve this requirement, we need to find some nonce n such that $H(d, n, p)$ begins with a certain number of 0s. By the properties of cryptographic hash functions, this is a hard problem. In fact, the more 0s we want t to begin with, the harder it becomes to find n.

This is the cryptographic puzzle that Bitcoin *miners* have to solve, in order to get to add a new block to the blockchain. The *work* that a miner performs is finding the right nonce n. The *proof* of work is the hash t which must begin with the required number of 0s. The difficulty of the puzzle can be adjusted by requiring more 0s or fewer 0s. The work (i.e., computational complexity) to find n is exponential in the ♠
required number of 0s. On the other hand, verifying the proof of work is simple: we just need to compute $H(d, n, p)$ and verify that it begins with the required number of 0s (and that it matches t).

What happens when a miner solves the puzzle? That miner is very happy! She is so happy that she lets every one know: she broadcasts to every node in the network the new block (d, n, p, t). By construction, $t = H(d, n, p)$, so the new block satisfies condition (1) of §8.4.1 by construction. The new block is appended to the blockchain (assuming it's an otherwise valid block, e.g., that d does not contain any invalid transactions) and the lucky miner gets a reward for her effort (e.g., some cryptocurrency is added to her account).

8.4.3 Allowing forks – the longest chain wins

Suppose two miners, X and Y, solve the puzzle at the same time or almost. Moreover, consider the scenario of §8.3.1 where X and Y have conflicting transactions, that is, the blocks mined by X and Y are different. Let B_X and B_Y be the blocks of X and Y, respectively.[43] What happens then? Some miners might choose to extend the chain from B_X, that is, to start mining a new block taking B_X as the parent block (in order to start mining, the hash p of the parent needs to be known, therefore, the miner needs to assume a given

[43]In fact, the blocks of X and Y will *always* be different because X's reward transaction will be included in B_X but not in B_Y, while Y's reward transaction will be included in B_Y but not in B_X.

parent). Other miners might go with B_Y.[44] This creates a *fork* in the chain,[45] that is, the chain is split into two branches, one starting with B_X, the other with B_Y. Eventually, new blocks will be added to each of the branches, and these branches will grow longer and longer. However, it is highly unlikely that the two branches will remain equal in length for long. This would require two miners to solve again the (two different) puzzles at about the same time, and again, and again. Eventually, one branch will become longer than the other. Miners will notice this and will abandon the shorter branch in favor of the longer one: this is what *longest-chain* consensus means. The longest branch will win the race.

Quoting from [115]:

> "Nodes always consider the longest chain to be the correct one and will keep working on extending it. If two nodes broadcast different versions of the next block simultaneously, some nodes may receive one or the other first. In that case, they work on the first one they received, but save the other branch in case it becomes longer. The tie will be broken when the next proof-of-work is found and one branch becomes longer; the nodes that were working on the other branch will then switch to the longer one."

Although consensus had already been studied for many years prior to 2008 when the Bitcoin white paper [115] came out (see §8.6), this way to do consensus was apparently novel and revolutionary. Here's how Prof. Tim Roughgarden puts it in his *Foundations of Blockchains* lecture videos (Lecture 8.1: *A Tale of Two Protocol Designs*): "The idea of allowing forks and resolving them within the protocol is a radical idea ... I'm not aware of any consensus protocol that took this approach prior to the Bitcoin white paper at the end of 2008." The lecture video is available here: `https://youtube.com/playlist?list=PLEGCF-WLh2RLOHv_xUGLqRts_9JxrckiA` .

Note that forks are part of the normal operation of blockchains, and can happen even when all nodes are honest and non-faulty, as in the scenario described above. Forks can also happen due to network delays. For example, consider a situation where there is some network outage resulting in a *network partition* where the network is split into two "islands" that cannot communicate with each other (during the outage). Again in this scenario, all nodes are honest and non-faulty, but the network is faulty. As a result, nodes in one island will continue solving puzzles and broadcasting blocks within their own island, and similarly in the other island. So the blockchain will again fork into two branches.

8.4.4 Refining the steps of §8.2.5

We are now in a position to refine the steps of block creation presented in the centralized view – §8.2.5. In the case of blockchains like Bitcoin (or the older version of Ethereum) that use longest-chain consensus with proof-of-work, the block creation protocol is as follows (see [115], and [2] under *Proof-of-work*, *Mining*):

1. A user broadcasts a digitally signed transaction request to all nodes (miners).
2. Each miner saves the transaction requests it hears about in a local pool.
3. From time to time, a miner selects a bunch of requests from its pool, for inclusion into a potential block.
4. The miner checks that all selected requests are correctly signed, syntactically valid, etc. The miner executes the selected transactions one by one (and discards those that revert). (In the case of old Ethereum the miner also updates their local EVM state according to each executed transaction.) The remaining set of selected, valid, ordered transactions roughly corresponds to the data field d in the formalization above, c.f. §8.4.1.

[44]Our two miners X and Y will probably go with their own blocks: X will continue with B_X and Y with B_Y. They have an incentive to do so, because they want to get their hardly earned reward! The one who eventually "wins the race" will win the lion's share of the reward. (The miner who loses the race might also get a smaller reward since, after all, they did the work – see [2], under *Proof-of-work*, *Mining*, *Ommer (uncle) blocks*.

[45]But not a *hard* fork like the Ethereum *DAO fork* or regular Ethereum upgrade forks: `https://ethereum.org/history/`.

5. The miner decides which parent block it should try to extend. Typically the block with the largest *height* will be chosen, i.e., the one corresponding to the longest chain. At this point, d and p of §8.4.1 are fixed. The miner can now start mining, that is, solving the proof-of-work puzzle, which is to find n such that $H(d, n, p)$ starts with a required number of 0s (§8.4.2).

6. When the miner finds such an n, the miner broadcasts the new block $(d, n, p, H(d, n, p))$ to all other miners.

7. Other miners hear about a new block (d, n, p, t). They verify that t begins with the required number of 0s and that $t = H(d, n, p)$. They also execute all transactions in d locally and verify that they are all valid (and in the case of old Ethereum update their local EVM state). The miners then remove all transactions in d from their pools and go back to step 2.

8.5 Proof-of-stake

In 2022, Ethereum switched from a PoW mechanism to a *proof-of-stake* (PoS) mechanism, "because it is more secure, less energy-intensive, and better for implementing new scaling solutions", according to the *Proof-of-stake* section of [2]. The same page claims that "proof-of-stake, as it is implemented on Ethereum, has been demonstrated to be more economically secure than proof-of-work", but without providing any evidence (e.g., citations to papers or references to studies) backing this claim.[46] Energy consumption, on the other hand, seems better researched: `https://ethereum.org/energy-consumption/` and `https://digiconomist.net/bitcoin-energy-consumption`. In-depth analyses of both PoW and PoS can be found in Prof. Tim Roughgarden's *Foundations of Blockchains* video series. PoW and PoS are compared in Lectures 12.23 and 12.24 of the series: `https://www.youtube.com/watch?v=XnS9VMqVPpE&list=PLEGCF-WLh2RLOHv_xUGLqRts_9JxrckiA&index=84`.

Before giving a high-level overview of Ethereum's PoS, let us remark that PoS is not by itself a consensus protocol, and for that matter, neither is PoW. Both PoW and PoS are mechanisms to determine the block *authors*, i.e., which node is allowed to append a new block to the chain – see *Sybil resistance & chain selection* in [2]. Both PoW and PoS are *Sybil-resistant*, i.e., they are designed to protect against Sibyl attacks (§8.3.2). But in a distributed blockchain we also need a mechanism to decide which chain wins when forks occur. This *chain selection rule* is a separate mechanism, and that's where consensus protocols come in. These somewhat separate choices are nicely described in *Lecture 9.1: Permissionless Consensus* of Prof. Tim Roughgarden's *Foundations of Blockchains* video series: see `https://youtu.be/rmmdAuc5i3g?list=PLEGCF-WLh2RLOHv_xUGLqRts_9JxrckiA&t=1818`.

Bitcoin uses PoW for Sybil-resistant block author selection, and longest-chain-wins as its chain selection rule.[47] Ethereum uses PoS for Sybil-resistant block author selection, and a protocol called *Gasper* which combines a fork choice algorithm with a mechanism to *finalize* blocks (see *Gasper* under the *Proof-of-stake* section of [2]). We briefly explain these concepts in what follows.[48]

8.5.1 Validators and time slots

Bitcoin nodes are called miners. Ethereum nodes are called *validators*. Just like a miner, a validator gets to create a new block from time to time, and broadcast it to all other validators. But unlike a miner, a validator does not have to solve a PoW puzzle. Instead, in Ethereum, time is divided in slots, and in every slot, one validator is chosen randomly to be the block creator for that slot. Now, the notion that time in Ethereum is divided into slots should raise some red flags in your head. Doesn't this mean that validators need to synchronize their clocks? Doesn't this require some sort of agreement, as we said in §8.3.1? Indeed, it does. My understanding is that Ethereum currently uses NTP, the standard time synchronization protocol ♠

[46] A youtube video is provided as reference in [2]: see *Proof-of-stake vs proof-of-work* under *Proof-of-stake*.

[47] The combination of PoW with longest-chain consensus is often called *Nakamoto consensus* [136].

[48] The complete Ethereum Proof-of-Stake Consensus Specifications can be found here: `https://github.com/ethereum/consensus-specs/tree/dev`.

of the internet, which is called "relatively centralized" in [39]. Relying on an external service such as NTP, especially one that might be "relatively centralized", can be problematic because of the risk for attacks [167].

Validators also check that blocks created by other validators are correctly formed. If that's indeed the case for a block, a validator broadcasts a vote (called an *attestation*) in favor of that block.

Why would a node want to become a validator? As a reward for operating Ethereum, validators collect the *priority fees* (also called *tips*) that users pay in order to incentivize validators to include a transaction in a block. What prevents validators from behaving maliciously? In order to become a validator, a user must deposit (*stake*) 32 ETH into a smart contract. If the validator is found to misbehave, some or all of their staked ETH can be destroyed (this is called *slashing*). Who detects misbehaving validators? Other validators! Validators that (correctly) catch others cheating receive a small "whistleblower" reward. As with other decisions (e.g., block creation), slashing is something that needs to be agreed upon among validators.

8.5.2 Finality, epochs, and attestations

Even in the case where all nodes are honest, blockchains can fork because of decentralization (§8.4.3). This is problematic for users submitting transactions to the blockchain; users now have worries such as *what if my transaction finds itself in the losing branch? what if the block that my transaction belongs to is reverted / rolled back? after how long can I be sure that this won't happen?* These are valid concerns. Therefore, blockchains need some kind of *finality* property which would guarantee that transactions (or the blocks they belong to) are considered *final* after some point.

Bitcoin does not have an explicit notion of finality. According to lecture 7 of [136], it is folklore in Bitcoin to assume that blocks older than the last 6 blocks are highly unlikely to be rolled back.[49]

In Ethereum, the notion of finality is explicit. According to the *Proof-of-stake* section of [2], a *transaction can be considered "finalized" if it has become part of a chain with a "supermajority link" between two checkpoints. Checkpoints occur at the start of each epoch [...].* An *epoch* equals 32 slots.[50] The first block in each epoch is a *checkpoint* block. What is so special about checkpoints? Validators *vote* for the last two checkpoints that they consider valid. If a pair of checkpoints (B_{n-1}, B_n) attracts votes representing at least ♠ two-thirds of the total staked ETH, then block B_{n-1} becomes *finalized*. The votes that validators cast every epoch are called *attestations*. Ethereum's fork choice algorithm works by identifying the fork that has the greatest weight of attestations in its history.

8.6 Theory of distributed systems

At the heart of blockchains are distributed protocols, the algorithms that the distributed nodes run to operate Bitcoin, Ethereum, etc. There is a rich theory behind such systems, with classic books such as [105], and more recent ones such as [151] which focuses specifically on blockchains. 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* [136], as well as a set of video lectures which we referred to above.

Consensus is perhaps the most fundamental problem in distributed systems. There are many variants of the problem, but the essence is common: a set of distributed nodes must reach some form of agreement, under some set of assumptions (regarding the network, type of node failures, attacks, etc). The need for consensus in practical distributed systems is prevalent. For instance, a distributed database of a bank must

[49]What exactly does it mean for a block to be rolled back? It means that the block is no longer part of the longest chain, and therefore the block is no longer considered valid. What happens to the transactions of that block in that case? These transactions are not considered valid either. As you might suspect, this could be problematic. For example, what if one such transaction is a payment from buyer A to seller B for a car? What if B already shipped the car to A? If the transaction is invalidated, then B no longer has A's payment, but A might still refuse to return the car! This problem could be solved by B delaying the shipment to A until enough time lapses to consider it very unlikely for the block to be rolled back. But what if B delays forever and never ships the goods to A? As stated in [115], "routine escrow mechanisms could easily be implemented to protect buyers".

[50]A slot equals 12 seconds therefore an epoch equals 6.4 minutes.

have a consistent view of the bank's accounts. [151] mentions aircraft control as another application, which in fact motivated original research in distributed fault tolerant computing.[51]

Consensus is a difficult problem, as attested by the sheer volume of the literature on the topic. It is both an old problem [125, 143, 105] as well as an active area of research (e.g., in the area of blockchains). Researchers in the theory of distributed systems investigate the fundamental properties of distributed protocols, such as safety and liveness properties (c.f. §6.12). For example, a safety property relevant for blockchains is **consistency**, namely, that the nodes agree on the block history.[52] A liveness property relevant for blockchains is that every valid transaction submitted by a user will eventually make it to the local view of all honest nodes. There are many consensus protocols, and it is not always easy to understand what their tradeoffs are in terms of both the properties that they guarantee as well as the assumptions that they make. It is not even easy to understand what is possible vs impossible to achieve (the theory of distributed systems has many impossibility results, e.g., see [102, 105]). In what follows, we give a partial overview of some key results.

8.6.1 The Byzantine Generals Problem

There are different versions of the consensus problem. A classic one is Byzantine agreement, captured by the famous Byzantine Generals Problem. Here's how the problem is described in the seminal paper [102] by Lamport et al.:[53]

> "We imagine that several divisions of the Byzantine army are camped outside an enemy city, each division commanded by its own general. The generals can communicate with one another only by messenger. After observing the enemy, they must decide upon a common plan of action. However, some of the generals may be traitors, trying to prevent the loyal generals from reaching agreement. The generals must have an algorithm to guarantee that
>
> A. All loyal generals decide upon the same plan of action. [...]
>
> B. A small number of traitors cannot cause the loyal generals to adopt a bad plan."

[102] shows that in general, $3n + 1$ nodes are needed to tolerate n Byzantine nodes (traitors), but if digital signatures are used, $2n + 1$ nodes are enough. Importantly, the solution assumes bounded-time message delivery, that is, that each message is delivered within some Δ time units. This is called the *synchronous model assumption.*

8.6.2 Impossibility results and tradeoffs: FLP and CAP

The synchronous model assumption in the solutions of [102] turns out to be necessary. The famous FLP impossibility theorem [69] shows that "no completely asynchronous consensus protocol can tolerate even a single unannounced process death." What FLP says is that consensus is impossible in the *asynchronous model* (where there is no bound on message transmission delays) even in the presence of a single crash

[51]Here's what Leslie Lamport says about this in https://www.microsoft.com/en-us/research/publication/sift-design-analysis-fault-tolerant-computer-aircraft-control/: "When it became clear that computers were going to be flying commercial aircraft, NASA began funding research to figure out how to make them reliable enough for the task. Part of that effort was the SIFT project at SRI. This project was perhaps most notable for producing the Byzantine generals problem and its solutions, first reported in [125]."

[52]Consistency does not require that all nodes have exactly the same local history view at all times. It is OK for a node to lag behind another node, for example, node X thinks that the history is B_0, B_1, B_2, B_3, whereas node Y thinks that the history is B_0, B_1, B_2. But it is not OK for X to think that the history is B_0, B_1, B_2, B_3 and Y to think that the history is B_0, B_1, B_2, B_4, for $B_4 \neq B_3$. Note that this definition of consistency does not allow forks!

[53]Leslie Lamport is a major figure in computer science. He has made fundamental contributions to many fields, including distributed systems and formal methods, which are both key topics in this course. Among many other contributions in formal methods, Lamport created the TLA+ framework (§6.4.4). Lamport has also made many contributions in the field of distributed systems, and it's worth reading some of his classic papers on the topic, e.g., [95, 102]. It's also worth reading some of Lamport's accounts about his papers, e.g.: https://www.microsoft.com/en-us/research/publication/byzantine-generals-problem/. Lamport won the Turing Award, the highest distinction in computer science, in 2013 – https://amturing.acm.org/award_winners/lamport_1205376.cfm.

fault (and no matter how many non-faulty nodes there are). Note that FLP considers the “easy” case of crash faults but no Byzantine faults (§8.3.2), and shows that even in that “easy” case, consensus cannot be achieved in general.

In computer science literature this type of results are called *impossibility results*. [102] also proves an impossibility result, namely, that if there are n traitors and $3n$ or fewer generals in total, then the Byzantine Generals Problem has no solution. In particular, if there are 3 generals in total and one of them is a traitor, the traitor can prevent agreement (in the sense of conditions A and B of §8.6.1). I wonder what this implies about airplanes that have 3 flight computers for redundancy.

Impossibility results do not spell doom. They spell tradeoffs. They tell us that we cannot have everything, and that we must choose what to prioritize. This is a big lesson not just for distributed systems but for any kind of system and for life in general. There are many types of tradeoffs in distributed system. A famous set ♠ of tradeoffs is set by the CAP principle: “Strong Consistency, High Availability, Partition-resilience: Pick at most 2” [71]. Safety vs liveness tradeoffs that different consensus protocols make are discussed in §8.6.4.

8.6.3 State Machine Replication in the permissioned setting

Another classic consensus problem is *state machine replication* (SMR) [95, 99, 143]. This is exactly the problem that concerns us. We have a state machine like EVM/Ethereum (§8.1) and we want to distribute (*replicate*) it. How to do that so that all replicas are in agreement (e.g., regarding the current state of the machine)? And how to achieve that in the presence of faults (crash or Byzantine) or even malicious attacks? And for real-world distributed systems which exhibit concurrency and asynchrony (§8.3.1)? In view of our discussion so far (§8.6.1-8.6.2) it should be clear that this is a really hard problem, and even impossible in general.

In a nutshell, protocols like Paxos [100] and Raft (`https://raft.github.io/`) solve SMR with crash faults in the *partially synchronous model* and provided the number of faulty nodes is $< 50\%$ (see lecture 5 of [136]). The partially synchronous model attempts to compromise between the synchronous model (all messages delivered with delay $\leq \Delta$) and the completely asynchronous model (no bound on the delay). The ♠ Tendermint protocol solves SMR with Byzantine faults in the partially synchronous model, provided the number of Byzantine nodes is $< 33\%$. Tendermint makes use of digital signatures (§9.2). Note that all these protocols operate in the permissioned setting (§8.3.3). Protocols like Tendermint are sometimes called BFT or PBFT protocols (see [137] or lecture 7 of [136]). (P)BFT stands for (Practical) Byzantine Fault Tolerance [41].

8.6.4 State Machine Replication in the permissionless setting

As discussed in §8.6.3, there exist several protocols for the SMR problem in the permissioned setting. But what about solving SMR in the permissionless setting, i.e., without assuming that all nodes are known in advance? This is really the problem that concerns us in this course, and this is the problem that blockchains like Bitcoin and Ethereum attempt to solve, using mechanisms such as those described in §8.4 and §8.5. In view of the difficulty of solving SMR (and consensus in general) even in the permissioned setting, one may ask *what are the theoretical guarantees that blockchains like Bitcoin and Ethereum provide?*

Answering this question in any depth is (way) beyond the scope of this class. But let us give part of the big picture. The following are taken almost verbatim from [137, 136]: Relative to BFT-type protocols like Tendermint, longest-chain consensus favors liveness over consistency. Nakamoto consensus loses the consistency safety property (see introduction to §8.6 above) even in the partially synchronous and permissioned setting, and with only honest nodes. This is because longest-chain consensus protocols allow forks. On the other hand, BFT-type protocols are designed with the idea that consistency must hold at all times, i.e., that forks never happen. As a result, if forks end up happening (e.g., due to implementation bugs, or because some design assumption breaks down, for instance, more than 33% Byzantine nodes) these forks are impossible to resolve without resorting to human coordination outside the protocol. And until the fork is resolved, BFT protocols give up liveness and *stall* (deadlock). In contrast, longest-chain protocols fork but don’t stall.

It is worth pointing out that traditional theoretical settings (including those of formal verification) are often quite rigid in the sense that they take a *worst-case scenario* approach. This is natural when trying to prove a theorem (e.g., an impossibility or complexity result). This worst-case approach is in contrast to the more realistic approach taken by real-world blockchains. For instance, the Ethereum documentation talks about its *crypto-economic security* in these terms: "Ethereum uses a proof-of-stake-based consensus mechanism that derives its crypto-economic security from a set of rewards and penalties applied to capital locked by stakers. This incentive structure encourages individual stakers to operate honest validators, punishes those who don't, and creates an extremely high cost to attack the network" – see the *Proof-of-stake* section of [2].

8.7 Scaling and Layer 2

Blockchains like Bitcoin and Ethereum are referred to as *Layer 1* (L1) networks (with *Layer 0* being the Internet). According to `https://ethereum.org/layer-2/learn/`: "The three desirable properties of a blockchain are that it is decentralized, secure, and scalable. The blockchain trilemma states that a simple blockchain architecture can only achieve two out of three. Want a secure and decentralized blockchain? You need to sacrifice scalability. This is where layer 2 networks come in. Ethereum has reached the network's current capacity with 1+ million transactions per day, with high demand for each of these transactions. The success of Ethereum and the demand to use it has caused gas prices to rise substantially. Therefore the need for scaling solutions has peaked as well." The goal of *Layer 2* (L2) networks is to improve scalability ♠ of L1s, essentially by offloading some of the burden away from the L1 blockchain. One approach for this is to use so-called *rollups*, which "bundle" many L2 transactions into a single L1 transaction. This raises the question *why should the L1 trust the transaction bundle that L2 proposes?* In *optimistic rollups* the L2 bundle is assumed to be valid by default; if there are suspicions of fraud, "a node can re-execute the transactions to check that they were executed honestly. If they uncover a discrepancy between the posted data and their own version they can post a cryptographic proof that demonstrates where some fraud took place" – `https://ethereum.org/developers/docs/scaling/optimistic-rollups/`. In *zero-knowledge rollups* the L2 proactively submits, in addition to the transaction bundle, a proof of its validity – `https://ethereum.org/developers/docs/scaling/zk-rollups/`.

We should remark that L2s are not the only approach to scaling. Other approaches are discussed in `https://ethereum.org/developers/docs/scaling/`.

9 Cryptography and Zero-Knowledge

Cryptographic tools such as hash functions and digital signatures are needed to ensure security and other properties of blockchains and smart contracts. In this section, we give a brief overview of some of these tools. We also touch upon the world of *zero-knowledge* (ZK), including interactive proofs, arguments, and ZK proofs [161]. These enable an untrusted party (e.g., an L2) to convince another party (e.g., a blockchain) that a certain computation has been performed correctly.

Hash functions and digital signatures are discussed in depth in cryptography textbooks such as [89]. Great resources on interactive proofs, arguments, and zero-knowledge are [161] and [33] (start with lectures 4 and 2). A curated list of papers, tools and other ZK resources can be found here: `https://github.com/StefanosChaliasos/Awesome-ZKP-Security`. For mathematical foundations, see [89] and [152].

9.1 Cryptographic hash functions

A *hash function* is a function from some set A (which could be very large or even infinite) to a finite set B. For example, let A be the set of all integers, and B the set $\{0, 1, ..., 7\}$. Then, a simple hash function $F : A \to B$ is the function that takes its input n, encodes it in binary, keeps only the 3 least significant bits, and outputs those back in decimal form. For example, $F(12) = 4$ (because 12 in binary is 1100, and 100 in decimal is 4), $F(15) = 7$ (because 15 in binary is 1111, and 111 in decimal is 7), $F(100) = 4$ (because 100 in binary is 1100100), $F(0) = 0$, and so on.

Our simple hash function F is unfortunately not good for security, because it's very predictable. For example, it can easily be *inverted*: if I asked you to find x such that $F(x) = 3$, you could easily do it (how?). Also, this function has many *collisions* (two inputs that yield the same output) and we can find such collisions easily (how?).

Security applications require *cryptographic* hash functions, which are very hard (practically impossible) to invert and for which it is very hard to find collisions. Canonical examples are the SHA-256 function – `https://en.wikipedia.org/wiki/SHA-2` – and the Keccak-256 function – `https://en.wikipedia.org/wiki/SHA-3`. To this date, no collisions have been found for either SHA-256 or Keccak-256.

9.2 Digital signatures

Digital signatures are, according to lecture 5 of [136] "one of the two most ubiquitous cryptographic primitives used in blockchain protocols" (the other being cryptographic hash functions). Digital signatures are used by senders to sign the transactions they submit to the blockchain (§8.1). Digital signatures may also be used by nodes running a consensus protocol to sign the messages they exchange with each other (§8.6.1 and §8.6.3).

A digital signature scheme allows say Alice to "sign" a message using her private key, so that anyone who knows her public key can verify that the message indeed came from Alice and was not modified in transit. Formally, a digital signature scheme consists of: (1) a way to generate public-private pairs of keys (e.g., `ssh-keygen`); (2) a way to sign messages, i.e., a function $F_{sign}(m, sk)$ which takes a message m and a private (secret) key sk and returns a *signature* (specific to m, as well as to sk); and (3) a way to verify signatures, i.e., a function $F_{verif}(m, s, pk)$ which takes a message m, a signature s, and a public key pk, and outputs 0 or 1. The idea is that when Alice wants to send a message m to Bob, she sends both m and $F_{sign}(m, sk_{Alice})$ (i.e., both the message and her signature *for that message*). When Bob receives a pair (m, s) from Alice, he checks whether $F_{verif}(m, s, pk_{Alice}) = 1$. If yes, Bob trusts that the message indeed came from Alice and has not been tampered with along the way.

Examples of digital signature schemes are RSA and ECDSA: `https://en.wikipedia.org/wiki/Digital_signature`. ECDSA relies on the computational hardness of the discrete logarithm problem.

9.3 Interactive proofs

An *interactive proof* (IP) generalizes the concept of classic proof. IPs are interesting on their own, but they are also stepping stones towards SNARKs or ZK proofs. An IP involves two parties: a *prover* P and a *verifier* V. P tries to convince V that $f(x) = y$, for some (computable) function f, and some x and y. For example, P might claim that y is the product of two (large) primes, x_1 and x_2; in this case, the input x is the pair (x_1, x_2) and f returns $x_1 \cdot x_2$. Or P might claim that $F(x) = y$, where F is a program that takes a lot of time/memory to run.

Now, the obvious solution to this problem is for P to simply send all the information (f, x, and y) to V, and let V check $f(x) = y$ for itself. This simple solution is already a (trivial) IP, but it has two main drawbacks. First, it is not ZK, because P reveals everything to V: for instance, what if P wants to keep (part of) x secret? Second, even if we don't care about ZK, we may not want P to have to send the entire x, as this might involve sending large amounts of data; or we may not V to have to compute f, as this might be a costly computation. All these cases are practically relevant in our setting. The case where we don't care about ZK but care about the cost of computation and communication is important in outsourcing computation from L1 to L2 chains (§8.7). The case where we do care about ZK is also important, for instance, in cases where some transactions are private. See lecture 2 of [33] for more.

Instead of using the trivial but unsatisfactory solution above, the idea behind IPs is that P and V engage in a finite interaction, i.e., an exchange of messages (hence the term *interactive* proof). During this interaction, V can challenge P in different ways. At the end of the exchange, V either accepts or rejects the claim that $f(x) = y$. Importantly, V can also appeal to randomness, e.g., toss coins when submitting challenges to P. An IP protocol must satisfy:

- **Completeness**: if $f(x) = y$ and both P and V are honest (i.e., they both follow the protocol) then V accepts.
- **Soundness**: if $f(x) \neq y$ then for any P' (therefore also for a possibly malicious P'), the probability that an honest V accepts when interacting with P' is less than a given ϵ.

Note that the soundness guarantee is generally probabilistic. That is, we allow a probability of error: a "cheating" prover might be able to convince an honest verifier that the claim is true even though it's not. But this happens only with probability at most ϵ, and in typical protocols we can make ϵ as small as we want, e.g., by increasing the number of messages exchanged in the protocol.

9.4 Arguments

Arguments are weaker versions of IPs. Arguments are weaker than IPs in the sense that arguments make stronger assumptions than IPs on the capabilities of the prover P. Specifically, an interactive argument is the same as an IP, but with a weaker soundness guarantee [161]:

- **Soundness**: if $f(x) \neq y$ then for any *polynomial-time* P', the probability that an honest V accepts when interacting with P' is less than a given ϵ.

Polynomial-time means that P' is only allowed to execute algorithms that run in polynomial time in the size of their inputs. This means that arguments offer weaker guarantees than IPs: in an argument, a computationally super-powerful (i.e., non-polynomial time) malicious prover could manage to cheat with probability $\geq \epsilon$. Whereas in an IP, even super-powerful provers can only cheat with probability $< \epsilon$.

9.5 SNARKs and zkSNARKs

If we have IPs, why would we care about arguments, which are weaker than IPs? Because arguments are a stepping stone to SNARKs: *Succint Non-interactive ARguments of Knowledge*. That's a mouthful, so let's break it down. Compared to IPs (and arguments), SNARKs give us: (1) succinctness, (2) non-interactivity, and (3) knowledge-soundness. The formal definitions are complex, so we only provide the intuition behind these properties. For the formal definitions see [161].

Non-interactivity means that instead of engaging in a protocol that involves the exchange of many messages, P only sends a single message (the *proof*) to V. Succinctness means that the proof that P sends to V is short (i.e., the message is small) and also easy to verify (i.e., V does not have to perform expensive computations in order to decide whether to accept or reject the proof). Knowledge-soundness means that V must be convinced that P "knows" a certain secret value used in the claim. For instance, V must be convinced that P knows w such that $C(x, w) = 0$, where C is an arithmetic circuit.

zkSNARKs are SNARKs which in addition to the three properties above also satisfy zero-knowledge.

9.6 Zero-Knowledge

Zero-knowledge is an additional property that IPs and arguments (including SNARKs) may or may not possess. A proof or argument is zero-knowledge if it protects P from a potentially malicious V by not revealing too much information to V. Here's a standard example, taken from [161]. Alice wants to login to some server. The server uses a cryptographic hash function H for authentication. That is, a user provides their password w, but the server is supposed to only store $H(w)$ (and not w itself). Then, each time a user wants to login, they provide w and the server checkes whether $H(w)$ equals the stored version, say s.

Now, suppose Alice does not trust the server: maybe she worries that the server might be malicious, and use her password to impersonate her; or maybe she worries that the server might be simply incompetent and prone to letting user passwords be leaked or stolen due to lack of sufficient security measures. Can Alice convince the server that she knows a w such that $H(w) = s$, but *without revealing* w? This is roughly what a ZK proof achieves.[54] From the point of view of Alice, the benefit is that she succeeds in logging in without revealing her password. On ther other hand, the server must also be confident that it is Alice who is trying to login, and not somebody else. What if Alice does not know w but she just guessed a w' (perhaps different from w) such that $H(w') = H(w)$ and therefore $H(w') = s$? We assume that Alice is not lucky enough to guess such a w', and also that she doesn't have the computational power to invert the cryptographic function H, i.e., to compute such a w'. Note that these assumptions are needed also in the non-ZK case: an honest server does not store w therefore cannot know whether Alice provides the real w or a w' which hashes to the same s as w. So, from the point of view of the server, a ZK proof that Alice knows *some* w' such that $H(w') = s$ without revealing w' is good enough to authenticate Alice.

The technical definitions of ZK are complex and beyond the scope of this document. See references provided at the beginning of §9. Here's a nice video which presents a basic ZK protocol for P to convince V that they have a 3-coloring of a graph: https://youtu.be/0tvcbw6k4eo. The video illustrates the three key elements of that ZK protocol, namely: (1) interactivity; (2) randomness (used both by V to unpredictably challenge the P, as well as by P to protect their knowledge of the coloring solution); and (3) "locking" (*commitment*) schemes.

10 Acknowledgments

This class is the evolution of *Special Topics* class CS 4973/6983 *Smart Contracts and Analysis* (SCA) which I taught in Fall 2025: https://www.ccs.neu.edu/~stavros/sca25.html. The idea for the SCA class came during my sabbatical in 2024-2025 at the University of Athens, Greece. I would like to thank Prof. Yannis Smaragdakis and his group for hosting me and teaching me about smart contracts. I would also like to thank the students in the SCA class.

References

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

[2] Ethereum documentation – consensus mechanisms. https://ethereum.org/developers/docs/consensus-mechanisms/.

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

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

[5] Isabelle. https://isabelle.in.tum.de/.

[6] Lean. https://lean-lang.org/.

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

[8] Rocq. https://rocq-prover.org/.

[9] Software Foundations. https://softwarefoundations.cis.upenn.edu/.

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

[54] At a high level this might sound like knowledge-soundness, but knowledge-soundness and zero-knowledge are different. Knowledge-soundness refers to the prover, and says that if P convinces V then P must know the secret w. Zero-knowledge refers to the verifier, and says that V does not gain any knowledge by interacting with P (other than the knowledge that P knows w). Technically, knowledge-soundness states that there exists an *extractor* algorithm which can be used to extract (i.e., compute) the secret w of P. If such an extractor algorithm exists, then P must know w. Zero-knowledge technically states that what V can compute after the interaction is equal to what V could have computed before the interaction. This can be captured using the notion of a *simulator* which simulates the interaction of P and V. Crucially, the simulator does not depend on P. Then, if an external observer cannot distinguish between the output of the simulator and the output of the real interaction between P and V, then this means that V gains as much knowledge from P as it would have gained from running the simulator.

[11] Z3. https://github.com/Z3Prover.

[12] Martín Abadi, Bruno Blanchet, and Cédric Fournet. The Applied Pi Calculus: Mobile Values, New Names, and Secure Communication. *J. ACM*, 65(1), October 2017.

[13] Martín Abadi and Andrew D. Gordon. A Calculus for Cryptographic Protocols: The Spi Calculus. *Information and Computation*, 148(1):1–70, 1999.

[14] Bowen Alpern and Fred B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181 – 185, 1985.

[15] Leonardo Alt, Martin Blicha, Antti E. J. Hyvärinen, and Natasha Sharygina. SolCMC: Solidity Compiler's Model Checker. In Sharon Shoham and Yakir Vizel, editors, *Computer Aided Verification*, Lecture Notes in Computer Science, pages 325–338. Springer, 2022.

[16] R. Alur, C. Courcoubetis, N. Halbwachs, T. Henzinger, P. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138:3–34, 1995.

[17] R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[18] Rajeev Alur. *Principles of Cyber-Physical Systems*. MIT Press, 2015.

[19] Medina Andresel, Cristinel Mateis, Dejan Nickovic, Spyridon Kounoupidis, Panagiotis Katsaros, and Stavros Tripakis. LTLGuard: Formalizing LTL Specifications with Compact Language Models and Lightweight Symbolic Reasoning, 2026.

[20] 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.

[21] 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.

[22] R-J. Back and J. Wright. *Refinement Calculus*. Springer, 1998.

[23] C. Baier, B. R. Haverkort, H. Hermanns, and J.-P. Katoen. Performance evaluation and model checking join forces. *Commun. ACM*, 53(9):76–85, September 2010.

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

[25] Christel Baier, Luca de Alfaro, Vojtěch Forejt, and Marta Kwiatkowska. Model checking probabilistic systems. In Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors, *Handbook of Model Checking*, pages 963–999. Springer, 2018.

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

[27] Clark Barrett. Satisfiability Modulo Theories. Slides presented at the Stanford DeFi Security Summit – https://docs.certora.com/en/latest/docs/user-guide/tutorials.html, August 30, 2022.

[28] Robert Beers. Pre-RTL formal verification: An Intel experience. In *Proceedings of the 45th ACM/IEEE Design Automation Conference*, pages 806–811, 2008.

[29] Yves Bertot and Pierre Casteran. *Interactive Theorem Proving and Program Development – Coq'Art: The Calculus of Inductive Constructions*. Springer, 2004.

[30] Dirk Beyer and Thomas Lemberger. Software Verification: Testing vs. Model Checking. In Ofer Strichman and Rachel Tzoref-Brill, editors, *Hardware and Software: Verification and Testing*, pages 99–114. Springer, 2017.

[31] Dirk Beyer and Thomas Lemberger. Six years later: testing vs. model checking. *Int. J. Softw. Tools Technol. Transfer*, 26:633–646, 2024.

[32] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu. Bounded model checking. *Advances in Computers*, 58:117–148, 2003.

[33] Dan Boneh, Shafi Goldwasser, Dawn Song, Justin Thaler, and Yupeng Zhang. Zero Knowledge Proofs – MOOC, Spring 2023. https://rdi.berkeley.edu/zk-learning/.

[34] A. R. Bradley and Z. Manna. *The calculus of computation - decision procedures with applications to verification*. Springer, 2007.

[35] Aaron R. Bradley. Sat-based model checking without unrolling. In *Proceedings of the 12th International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI'11, pages 70–87, Berlin, Heidelberg, 2011. Springer.

[36] Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner, editors. *Model-Based Testing of Reactive Systems*. Springer, 2005.

[37] J. Burch, E. Clarke, D. Dill, L. Hwang, and K. McMillan. Symbolic model checking: 10^{20} states and beyond. In *5th LICS*, pages 428–439. IEEE, 1990.

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

[39] Vitalik Buterin. Network-adjusted timestamps. https://ethresear.ch/t/network-adjusted-timestamps/4187, Nov 2018.

[40] Igor Buzhinsky. Formalization of natural language requirements into temporal logics: a survey. In *17th IEEE International Conference on Industrial Informatics, INDIN 2019, Helsinki, Finland, July 22-25, 2019*, pages 400–406. IEEE, 2019.

[41] Miguel Castro and Barbara Liskov. Practical Byzantine Fault Tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, pages 173–186. USENIX Association, 1999.

[42] Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. The nuXmv Symbolic Model Checker. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification (CAV)*, pages 334–342. Springer, 2014.

[43] Stefanos Chaliasos, Jens Ernstberger, David Theodore, David Wong, Mohammad Jahanara, and Benjamin Livshits. SoK: What don't we know? Understanding Security Vulnerabilities in SNARKs. https://arxiv.org/abs/2402.15293, 2024.

[44] Rod Chapman, Adam Petcher, Torben Hansen, Yan Peng, Tancrède Lepoint, Cameron Bytheway, and Panos Kampanakis. Formal verification of cryptographic software at aws - current practices and future trends. 2024.

[45] Alonzo Church. Applications of recursive arithmetic to the problem of circuit synthesis. In *Summaries of the Summer Institute of Symbolic Logic*, volume 1, pages 3–50, 1957.

[46] E. Clarke, E. Emerson, S. Jha, and A. Sistla. Symmetry reductions in model checking. In *CAV'98*, pages 147–158. Springer, 1998.

[47] Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors. *Handbook of Model Checking*. Springer, 2018.

[48] Michael R. Clarkson and Fred B. Schneider. Hyperproperties. In *21st IEEE Computer Security Foundations Symposium*, pages 51–65, 2008.

[49] Matt Condon. Parity Wallet Hack 2: Electric Boogaloo. Available at: https://hackernoon.com/parity-wallet-hack-2-electric-boogaloo-e493f2365303, November 8th, 2017.

[50] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th ACM Symp. POPL*, 1977.

[51] 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.

[52] Richard A. De Millo, Richard J. Lipton, and Alan J. Perlis. Social processes and proofs of theorems and programs. *Commun. ACM*, 22(5):271–280, May 1979.

[53] Leonardo de Moura and Sebastian Ullrich. The Lean 4 Theorem Prover and Programming Language. In André Platzer and Geoff Sutcliffe, editors, *Automated Deduction - CADE 28 - 28th International Conference on Automated Deduction, Virtual Event, July 12-15, 2021, Proceedings*, volume 12699 of *Lecture Notes in Computer Science*, pages 625–635. Springer, 2021.

[54] W.P. de Roever, H. Langmaack, and A. Pnueli (Eds.). *Compositionality: The Significant Difference.* LNCS. Springer, 1998.

[55] 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.

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

[57] 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.

[58] Giorgio Dell’Immagine. Introducing clean, a formal verification DSL for ZK circuits in Lean4. https://blog.zksecurity.xyz/posts/clean/, Mar 27, 2025.

[59] 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.

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

[61] 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.

[62] John R. Douceur. The Sybil Attack. In Peter Druschel, Frans Kaashoek, and Antony Rowstron, editors, *Peer-to-Peer Systems*, pages 251–260. Springer, 2002.

[63] A. Doxiadis, C.H. Papadimitriou, A. Papadatos, and A. Di Donna. *Logicomix: An Epic Search for Truth.* 2009.

[64] Derek Egolf and Stavros Tripakis. Accelerating Protocol Synthesis and Detecting Unrealizability with Interpretation Reduction. In *31st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2025.

[65] E. Emerson and C. Lei. Modalities for model checking: Branching time logic strikes back. In *12th ACM Symp. POPL*, 1985.

[66] Josselin Feist, Jay Little, and Andy Ying. Parity Security Assessment. Trail of Bits audit report, available at https://www.trailofbits.com/documents/parity.pdf, July 22, 2018.

[67] 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.

[68] Bernd Finkbeiner. Synthesis of reactive systems. In Javier Esparza, Orna Grumberg, and Salomon Sickert, editors, *Dependable Software Systems Engineering*, volume 45 of *NATO Science for Peace and Security Series, D: Information and Communication Security*, pages 72–98. IOS Press, 2016.

[69] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *J. ACM*, 32(2):374–382, April 1985.

[70] R.W. Floyd. Assigning meanings to programs. In *In. Proc. Symp. on Appl. Math. 19*, pages 19–32. American Mathematical Society, 1967.

[71] Armando Fox and Eric A. Brewer. Harvest, yield, and scalable tolerant systems. In *Proceedings of the The Seventh Workshop on Hot Topics in Operating Systems*, HOTOS '99, page 174, USA, 1999. IEEE Computer Society.

[72] Hubert Garavel and Frédéric Lang. Equivalence Checking 40 Years After: A Review of Bisimulation Tools. In Nils Jansen, Mariëlle Stoelinga, and Petra van den Bos, editors, *A Journey from Process Algebra via Timed Automata to Model Learning : Essays Dedicated to Frits Vaandrager on the Occasion of His 60th Birthday*, pages 213–265. Springer, 2022.

[73] Hubert Garavel, Maurice H. ter Beek, and Jaco van de Pol. The 2020 Expert Survey on Formal Methods. In *Formal Methods for Industrial Critical Systems: 25th International Conference, FMICS 2020, Vienna, Austria, September 2–3, 2020, Proceedings*, page 3–69, Berlin, Heidelberg, 2020. Springer-Verlag.

[74] Markella Gioka. Smart Contracts to Embeddings: Using off-the-shelf LLMs for Fun and Profit. Video: https://www.youtube.com/watch?v=DmfUBh75F1E.

[75] P. Godefroid and P. Wolper. Using partial orders for the efficient verification of deadlock freedom and safety properties. In *4th CAV*, July 1991.

[76] Harrison Goldstein, Joseph W. Cutler, Daniel Dickstein, Benjamin C. Pierce, and Andrew Head. Property-Based Testing in Practice. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, New York, NY, USA, 2024. Association for Computing Machinery.

[77] 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.

[78] Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. Program synthesis. *Foundations and Trends in Programming Languages*, 4(1-2):1–119, 2017.

[79] A. Finn Hackett, Joshua Rowe, and Markus Alexander Kuppe. Understanding Inconsistency in Azure Cosmos DB with TLA+. In *45th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice, SEIP@ICSE 2023, Melbourne, Australia, May 14-20, 2023*, pages 1–12. IEEE, 2023.

[80] John Harrison. *Handbook of Practical Logic and Automated Reasoning.* Cambridge University Press, 2009.

[81] Klaus Havelund and Grigore Roşu. An Overview of the Runtime Verification Tool Java PathExplorer. *Form. Methods Syst. Des.*, 24(2):189–215, March 2004.

[82] Jie He, Ezio Bartocci, Dejan Ničković, Haris Isakovic, and Radu Grosu. DeepSTL: from english requirements to signal temporal logic. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, pages 610–622, New York, NY, USA, 2022. Association for Computing Machinery.

[83] C. A. R. Hoare. An axiomatic basis for computer programming. *Comm. ACM*, 12(10):576–580, 1969.

[84] Gerard J. Holzmann. *Design and Validation of Computer Protocols.* Prentice Hall, 1991.

[85] Gerard J. Holzmann. An analysis of bitstate hashing. In *Formal Methods in System Design*, pages 301–314. Chapman & Hall, 1998.

[86] Gerard J. Holzmann. *The Spin Model Checker.* Addison-Wesley, 2003.

[87] Gerard J. Holzmann. Formalizing Requirements Is $\Diamond\Box$ Hard. In Ezio Bartocci, Rance Cleaveland, Radu Grosu, and Oleg Sokolsky, editors, *From Reactive Systems to Cyber-Physical Systems - Essays Dedicated to Scott A. Smolka on the Occasion of His 65th Birthday*, volume 11500 of *Lecture Notes in Computer Science*, pages 51–56. Springer, 2019.

[88] Tengyun Jiao, Zhiyu Xu, Minfeng Qi, Sheng Wen, Yang Xiang, and Gary Nan. A Survey of Ethereum Smart Contract Security: Attacks and Detection. *Distrib. Ledger Technol.*, 3(3), September 2024.

[89] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography (2nd edition).* CRC Press, 2015.

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

[91] 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.

[92] Igor Konnov, Markus Kuppe, and Stephan Merz. Specification and Verification with the TLA+ Trifecta: TLC, Apalache, and TLAPS. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Verification Principles - 11th International Symposium, ISoLA 2022, Rhodes, Greece, October 22-30, 2022, Proceedings, Part I*, volume 13701 of *Lecture Notes in Computer Science*, pages 88–105. Springer, 2022.

[93] Moez Krichen and Stavros Tripakis. Conformance Testing for Real-Time Systems. *Formal Methods in System Design*, 34(3):238–304, June 2009.

[94] Daniel Kroening and Ofer Strichman. *Decision Procedures - An Algorithmic Point of View.* Texts in Theoretical Computer Science. An EATCS Series. Springer, 2008.

[95] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM*, 21(7):558–565, 1978.

[96] L. Lamport. Sometimes is sometimes “not never”– on the temporal logic of programs. In *7th ACM Symp. POPL*, pages 174–185, 1980.

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

[98] Leslie Lamport. PlusCal Tutorial. https://lamport.azurewebsites.net/tla/learning.html and https://lamport.azurewebsites.net/tla/tutorial/intro.html.

[99] Leslie Lamport. The Implementation of Reliable Distributed Multiprocess Systems. *Computer Networks*, 2:95–114, August 1978.

[100] Leslie Lamport. The Part-Time Parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[101] 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.

[102] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine Generals Problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, July 1982.

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

[104] H. Lewis and C. Papadimitriou. *Elements of the Theory of Computation, 2/e*. Prentice-Hall, 1997.

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

[106] Sharad Malik. The Quest for Efficient Boolean Satisfiability Solvers (slides). https://www.princeton.edu/~sharad/CMUSATSeminar.pdf.

[107] Sharad Malik and Lintao Zhang. Boolean satisfiability: From theoretical hardness to practical success. *Communications of the ACM*, 52(8):76–82, 2009.

[108] Valentin J.M. Manés, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. The Art, Science, and Engineering of Fuzzing: A Survey. *IEEE Transactions on Software Engineering*, 47(11):2312–2331, 2021.

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

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

[111] Z. Manna and A. Pnueli. Temporal Verification of Reactive Systems: Progress. Available at https://theory.stanford.edu/~zm/tvors3.html, 2004.

[112] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[113] Paul Molitor, Janett Mohnke, Bernd Becker, and Christoph Scholl. *Equivalence Checking of Digital Circuits – Fundamentals, Principles, Methods*. Springer, 2004.

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

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

[116] C. Newcombe, T. Rath, F. Zhang, B. Munteanu, M. Brooker, and M. Deardeuff. How Amazon Web Services Uses Formal Methods. *Commun. ACM*, 58(4):66–73, March 2015.

[117] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL – A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

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

[119] 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.

[120] Rodrigo Otoni, Igor Konnov, Jure Kukovec, Patrick Eugster, and Natasha Sharygina. Symbolic model checking for TLA+ made faster. In Sriram Sankaranarayanan and Natasha Sharygina, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 29th International Conference, TACAS*, volume 13993 of *Lecture Notes in Computer Science*, pages 126–144. Springer, 2023.

[121] Rodrigo Otoni, Matteo Marescotti, Leonardo Alt, Patrick Eugster, Antti E. J. Hyvärinen, and Natasha Sharygina. A solicitous approach to smart contract verification. *ACM Trans. Priv. Secur.*, 26(2):15:1–15:28, 2023.

[122] Santiago Palladino. The Parity Wallet Hack Explained. Available at: https://blog.openzeppelin.com/on-the-parity-wallet-multisig-hack-405a8c12e8f7, July 19, 2017.

[123] James Parker. zkLean: A DSL for ZK statement verification. https://www.galois.com/articles/zklean-a-dsl-for-zk-statement-verification, March 16, 2026.

[124] David Lorge Parnas. "Formal methods" technology transfer will fail. *Journal of Systems and Software*, 40(3):195–198, 1998. Formal Methods Technology Transfer.

[125] Marshall Pease, Robert Shostak, and Leslie Lamport. Reaching Agreement in the Presence of Faults. *Journal of the Association for Computing Machinery 27*, 2, April 1980. 2005 Edsger W. Dijkstra Prize in Distributed Computing.

[126] Sergey Petrov. Another Parity Wallet hack explained. Available at: https://medium.com/@Pr0Ger/another-parity-wallet-hack-explained-847ca46a2e1c, Nov 7, 2017.

[127] B. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[128] Nir Piterman and Amir Pnueli. Temporal logic and fair discrete systems. In Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors, *Handbook of Model Checking*, pages 27–73. Springer, 2018.

[129] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, SFCS '77, pages 46–57, 1977.

[130] Amir Pnueli and Roni Rosner. On the synthesis of a reactive module. In *ACM Symp. POPL*, 1989.

[131] Viorel Preoteasa, Iulia Dragomir, and Stavros Tripakis. The Refinement Calculus of Reactive Systems. *Information and Computation*, 285, 2022.

[132] Haseeb Qureshi. A hacker stole $31M of Ether — how it happened, and what it means for Ethereum. Available at: https://www.freecodecamp.org/news/a-hacker-stole-31m-of-ether-how-it-happened-and-what-it-means-for-ethereum-9e5dc29e33ce, July 20, 2017.

[133] Amirhosein Rajabi. Towards formally verifying the TLS 1.3 Key Schedule Security in SSBee. Master's thesis, Aalto University, 2025.

[134] P. Ramadge and W. Wonham. The control of discrete event systems. *Proceedings of the IEEE*, January 1989.

[135] 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.

[136] Tim Roughgarden. COMS 4995-001: The Science of Blockchains, Spring 2025. `https://timroughgarden.org/s25/`.

[137] Tim Roughgarden. Foundations of Blockchains Lectures #8: Longest-Chain Consensus (ROUGH DRAFT). `https://timroughgarden.github.io/fob21/l/l8.pdf`, 2021-2022.

[138] John Rushby. Formal Methods and the Certification of Critical Systems. Technical Report CSL-93-7, Computer Science Laboratory, SRI International, December 1993. Available at `http://www.csl.sri.com/~rushby/papers/csl-93-7.pdf`.

[139] John Rushby. The Interpretation and Evaluation of Assurance Cases. Technical Report SRI-CSL-15-01, Computer Science Laboratory, SRI International, Menlo Park, CA, July 2015. Available at `http://www.csl.sri.com/users/rushby/papers/sri-csl-15-1-assurance-cases.pdf`.

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

[141] 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.

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

[143] Fred B. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Comput. Surv.*, 22(4):299–319, December 1990.

[144] William Schultz, Edward Ashton, Heidi Howard, and Stavros Tripakis. Scalable, Interpretable Distributed Protocol Verification by Inductive Proof Slicing. arXiv eprint 2404.18048, 2024.

[145] William Schultz, Edward Ashton, Heidi Howard, and Stavros Tripakis. Interactive Safety Verification of Distributed Protocols by Inductive Proof Decomposition. In *18th NASA Formal Methods Symposium*, 2026.

[146] 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.

[147] 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.

[148] 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.

[149] 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.

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

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

[152] Victor Shoup. *A Computational Introduction to Number Theory and Algebra*. 2008.

[153] Malte Skarupke. Using TLA+ to Fix a Very Difficult glibc Bug. Talk at C++Now 2025: https://youtu.be/Brgfp7_OP2c?si=1jKLQurEMDRXtN6d, 2025.

[154] 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.

[155] 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.

[156] 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. (OOPSLA)*, 5:1–30, October 2021.

[157] 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.

[158] Sebastian Stock, Jannik Dunkelau, and Atif Mashkoor. Application of AI to formal methods - an analysis of current trends. https://arxiv.org/abs/2411.14870, 2025.

[159] Yuqiang Sun, Daoyuan Wu, Yue Xue, Han Liu, Haijun Wang, Zhengzi Xu, Xiaofei Xie, and Yang Liu. Gptscan: Detecting logic vulnerabilities in smart contracts by combining gpt with program analysis. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, New York, NY, USA, 2024. Association for Computing Machinery.

[160] Ertem Nusret Tas, David Tse, Fangyu Gai, Sreeram Kannan, Mohammad Ali Maddah-Ali, and Fisher Yu. Bitcoin-enhanced proof-of-stake security: Possibilities and impossibilities, 2025.

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

[162] Stavros Tripakis. A simple way to explain undecidability. https://hal.science/hal-02068449, 14 Mar 2019.

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

[164] Stavros Tripakis. CS 4973/6983 Smart Contracts and Analysis – Lecture Notes, Fall 2025. Available at https://www.ccs.neu.edu/~stavros/lectures.pdf, 2025.

[165] A. M. Turing. On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 1936.

[166] A. Valmari. Eliminating redundant interleavings during concurrent program verification. In *PARLE'89*, volume 366 of *LNCS*, pages 89–103. Springer, 1989.

[167] Alex Vlasov and Lu Yu. Why clock sync matters in Ethereum 2.0. https://hackmd.io/@ericsson49/BJfLjEX-8, Aug 8, 2022.

[168] Fabian Vogelsteller and Vitalik Buterin. ERC-20: Token Standard. https://eips.ethereum.org/EIPS/eip-20, 2015-11-19.

[169] Scott Wesley, Maria Christakis, Jorge A. Navas, Richard Trefler, Valentin Wüstholz, and Arie Gurfinkel. Verifying Solidity Smart Contracts via Communication Abstraction in SmartACE. In Bernd Finkbeiner and Thomas Wies, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 425–449. Springer, 2022.

[170] Gavin Wood. Ethereum: A Secure Decentralized Generalized Transaction Ledger – Shanghai Version efc5f9a 2025-02-04. Available at https://ethereum.github.io/yellowpaper/paper.pdf, 2025.

[171] Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John Fitzgerald. Formal Methods: Practice and Experience. *ACM Comput. Surv.*, 41(4), October 2009.

[172] Zhuo Zhang, Brian Zhang, Wen Xu, and Zhiqiang Lin. Demystifying exploitable bugs in smart contracts. In *Proceedings of the 45th International Conference on Software Engineering*, ICSE '23, page 615–627. IEEE Press, 2023.

[173] 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.