10

Smart Contracts

This chapter introduces smart contracts. This concept is not new; however, with the advent of blockchain technology, interest in this idea has revived. Smart contracts are now an ongoing and intense area of research in the blockchain space. Many blockchains have emerged that support smart contracts.

Due to benefits such as the increased security, cost-saving, and transparency that smart contracts can bring to many industries (especially the finance industry), rigorous research is in progress at various commercial and academic institutions to make the implementation of smart contracts easier, more practical, business-friendly, and more secure as soon as possible.

In this chapter, we will explore the ideas that led to the development of smart contracts, including its relevant history and key definitions. We will also cover concepts like the deployment of smart contracts, notable security aspects of smart contracts, and, finally, oracles. First, we will look at the history of smart contracts.

History

Nick Szabo first theorized smart contracts in the 1990s, in an article called *Formalizing and Securing Relationships on Public Networks*. This theory was presented almost 20 years before the real potential and benefits of smart contracts were appreciated, that is, before the invention of Bitcoin and the subsequent development of other more advanced blockchain platforms, such as Ethereum.

Smart contracts are described by Szabo as follows:

> *"A smart contract is an electronic transaction protocol that executes the terms of a contract. The general objectives are to satisfy common contractual conditions (such as payment terms, liens, confidentiality, and even enforcement), minimize exceptions both malicious and accidental, and minimize the need for trusted intermediaries. Related economic goals include lowering fraud loss, arbitrations and enforcement costs, and other transaction costs."*

The original article that was written by Szabo is available at `http://firstmonday.org/ojs/index.php/fm/article/view/548`.

Smart contract functionality was implemented in a limited fashion in Bitcoin in 2009. Bitcoin supports a restricted scripting language called **script**, which allows the transfer of bitcoins between users. However, this is not a Turing complete language and does not support arbitrary program development. It can be regarded as a limited function calculator with only simple arithmetic operations, whereas smart contracts can be considered general-purpose computers that support writing any program.

Definition

There are many definitions of smart contracts. An online search reveals many definitions. However, while the definitions provided online at various sources are correct and useful, in my opinion, they are not complete. It is crucial to properly and completely define what a smart contract is. The following is my attempt to provide a comprehensive generalized definition of a smart contract:

A smart contract is a secure and unstoppable computer program representing an agreement that is automatically executable and enforceable.

Dissecting this definition reveals that a smart contract is, fundamentally, a computer program that is written in a language that a computer or target machine can understand. Also, it encompasses agreements between parties in the form of business logic. Another fundamental idea is that smart contracts are automatically executed according to the instruction that is coded in, for example, when certain conditions satisfy. They are enforceable, which means that all contractual terms perform as specified and expected, even in the presence of adversaries.

Enforcement is a broader term that encompasses traditional enforcement in the form of a law, along with the implementation of specific measures and controls that make it possible to execute contract terms without requiring any intervention.

Preferably, smart contracts should not rely on any traditional methods of enforcement. Instead, they should work on the principle that *code is the law*, which means that there is no need for an arbitrator or a third party to enforce, control, or influence the execution of a smart contract. Smart contracts are self-enforcing as opposed to legally enforceable. This idea may sound like a libertarian's dream, but it is entirely possible and is in line with the true spirit of smart contracts.

Moreover, they are secure and unstoppable, which means that these computer programs are fault-tolerant and executable in a reasonable (finite) amount of time. These programs should be able to execute and maintain a healthy internal state, even if external factors are unfavorable. For example, imagine a typical computer program that is encoded with some logic and executes according to the instruction coded within it. However, if the environment it is running in or the external factors it relies on deviate from the usual or expected state, the program may react arbitrarily or abort. Smart contracts must be immune to this type of issue.

> Blockchain platforms play a vital role in providing the necessary underlying network with security guarantees required to run the smart contracts.

In some scenarios, security and unstoppability may well be considered optional features. Still, it will provide more significant benefits in the long run if security and unstoppable properties are included in the smart contract definition. This inclusion will allow researchers to focus on these aspects from the start and will help to build strong foundations on which further research can then be based. There is also a suggestion by some researchers that smart contracts do not need to be automatically executable; instead, they can be what's called automatable, due to the manual human input required in some scenarios. For example, the manual verification of a medical record might be needed by a qualified medical professional. In such cases, fully automated approaches may not work best. While it is true that, in some instances, human input and control are helpful, they are not necessary. For a contract to be truly smart, in my opinion, it has to be fully automated. Certain inputs that need to be provided by people can and should also be automated. Oracles can be used for this purpose. We will discuss oracles in more detail later on in this chapter.

Smart contracts usually operate by managing their internal state using a state machine model provided by the underlying blockchain. This allows the development of a practical framework for programming smart contracts, where the state of a smart contract is advanced further based on some predefined criteria and conditions.

There is also an ongoing debate on whether computer code is acceptable as a conventional contract in a court of law. A smart contract is different in presentation from traditional legal prose, albeit they do represent and enforce all required contractual clauses. Still, a court of law does not understand computer code. This dilemma raises several questions about how a smart contract can be legally binding: can it be developed in such a way that it is readily acceptable and understandable in a court of law? Is it possible for dispute resolution be implemented within the code? Moreover, regulatory and compliance requirements are other topics that require addressing before smart contracts can become as efficient as traditional legal documents.

The legality of smart contracts is uncertain in many jurisdictions. Still, a recent exciting development in this space made crypto assets and smart contracts valid in English law by recognizing crypto assets as tradeable property and smart contracts as enforceable agreements. This announcement was made by the **UK Jurisdiction Taskforce** (**UKJT**) of the Lawtech Delivery Panel. More information about this, including the full legal statement, is available here: `https://technation.io/news/uk-takes-significant-step-in-legal-certainty-for-smart-contracts-and-cryptocurrencies/`.

Even though smart contracts are named smart, they only do what they have been programmed to do. This very property of smart contracts ensures that smart contracts produce the same output every time they are executed. The deterministic nature of smart contracts is highly desirable in blockchain platforms due to consistency requirements.

Now, this gives rise to a problem whereby a large gap between the real world and the blockchain world emerges. In this situation, natural language is not understood by the smart contract, and, similarly, computer code is not acceptable for the natural world. So, a few questions arise: how can real-life contracts be deployed on a blockchain? How can this bridge between the real world and the smart contract world be built?

These questions open up various possibilities, such as making smart contract code readable not only by machines but also by people. If humans and machines can both understand the code written in a smart contract, it might become acceptable in legal situations, as opposed to just a piece of code that no one understands except for programmers. This desirable property is an area that is ripe for research, and a significant research effort has been expended in this area to answer questions around the semantics, meaning, and interpretation of smart contracts.

Some work has already been done to describe natural language contracts formally, by combining both smart contract code and natural language contracts through linking contract terms with machine-understandable elements. This is achieved using a markup language called the **Legal Knowledge Interchange Format** (**LKIF**), which is an XML schema for representing theories and proofs. It was developed under the Estrella project in 2008.

More information is available in the research paper at `https://doi.org/10.1007/978-3-642-15402-7_30`.

Further details can be found here: `http://www.estrellaproject.org/?page_id=5`.

Another requirement regarding the properties of smart contracts is that they must be deterministic. This property will allow a smart contract to be run by any node on a network and achieve the same result. If the result differs even slightly between nodes, then a consensus cannot be reached, and a whole paradigm of distributed consensuses on the blockchain can fail. Moreover, it is also desirable that the contract language itself is deterministic, thus ensuring the integrity and stability of the smart contracts. A deterministic property means that for the same input, there is always the same output. In other words, the system must not manifest different behavior for the same input in different runs. In a blockchain network, this would mean that smart contracts developed using the smart contract programming language do not produce different results on different nodes.

Let's take, for example, various floating-point operations calculated by various functions in a variety of programming languages that can produce different results in different runtime environments. Another example is some math functions in JavaScript, which can produce different results for the same input on different browsers and can, in turn, lead to various bugs. This scenario is unacceptable in smart contracts because if the results are inconsistent between the nodes, then a consensus will never be achieved.

The deterministic feature ensures that smart contracts always produce the same output for a specific input. In other words, programs, when executed, produce a reliable and accurate business logic that is entirely in line with the requirements programmed in the high-level code.

In summary, a smart contract has the following properties:

- **Automatically executable**: It is self-executable on a blockchain without requiring any intervention.
- **Enforceable**: This means that all contract conditions are enforced automatically.
- **Secure**: This means that smart contracts are tamper-proof (or tamper-resistant) and run with security guarantees. The underlying blockchain usually provides these security guarantees; however, the smart contract programing language and the smart contract code themselves must be correct, valid, and verified.
- **Deterministic**: The deterministic feature ensures that smart contracts always produce the same output for a specific input. Even though it can be considered to be part of the secure property, defining it here separately ensures that the deterministic property is considered one of the important properties.
- **Semantically sound**: This means that they are complete and meaningful to both people and computers.
- **Unstoppable**: This means that adversaries or unfavorable conditions cannot negatively affect the execution of a smart contract. When the smart contracts execute, they complete their performance deterministically in a finite amount of time.

It could be argued that the first four properties are required as a minimum, whereas the latter two may not be necessary or applicable in some scenarios and can be relaxed. For example, a financial derivatives contract does not, perhaps, need to be semantically sound and unstoppable but should at least be automatically executable, enforceable, deterministic, and secure. On the other hand, a title deed needs to be semantically sound and complete; therefore, for it to be implemented as a smart contract, the language that it is written in must be understood by both computers and people.

Ian Grigg addressed this issue of interpretation with his invention of Ricardian contracts, which we will introduce in the next section.

Ricardian contracts

Ricardian contracts were initially proposed in the paper, *Financial Cryptography in 7 Layers*, by *Ian Grigg*, in the late 1990s.

This paper is available at `https://iang.org/papers/fc7.html`.

Ricardian contracts were initially used in a bond trading and payment system called **Ricardo**. The fundamental idea behind this contract is to write a document that is understood and accepted by both a court of law and computer software. Ricardian contracts address the challenge of the issuance of value over the internet. A Ricardian contract identifies the issuer and captures all the terms and clauses of the contract in a document to make it acceptable as a legally binding contract.

A Ricardian contract is a document that has several of the following properties:

- It is a contract offered by an issuer to holders
- It is a valuable right held by holders and managed by the issuer
- It can be easily read by people (like a contract on paper)
- It can be read by programs (parsable, like a database)
- It is digitally signed
- It carries the keys and server information
- It is allied with a unique and secure identifier

> The preceding information is based on the original definition by Ian Grigg at `http://iang.org/papers/ricardian_contract.html`.

In practice, the contracts are implemented by producing a single document that contains the terms of the contract in legal prose and the required machine-readable tags. This document is digitally signed by the issuer using their private key. This document is then hashed using a message digest function to produce a hash by which the document can be identified. This hash is then further used and signed by parties during the performance of the contract to link each transaction with the identifier hash, therefore serving as evidence of intent. This is depicted in the next diagram and is usually called a bowtie model.

The diagram shows a number of elements:

- The **World of Law** is on the left-hand side from where the document originates. This document is a written contract in legal prose with some machine-readable tags.
- This document is then hashed.
- The resultant message digest is used as an identifier throughout the **World of Accountancy**, as shown on the right-hand side of the diagram.

The **World of Accountancy** element represents any accounting, trading, and information systems that are being used in the business to perform various business operations. The idea behind this flow is that the message digest generated by hashing the document is first used in a so-called **genesis transaction**, or first transaction, and then it is used in every transaction as an identifier throughout the operational execution of the contract. This way, a secure link is created between the original written contract and every transaction in the **World of Accounting**:

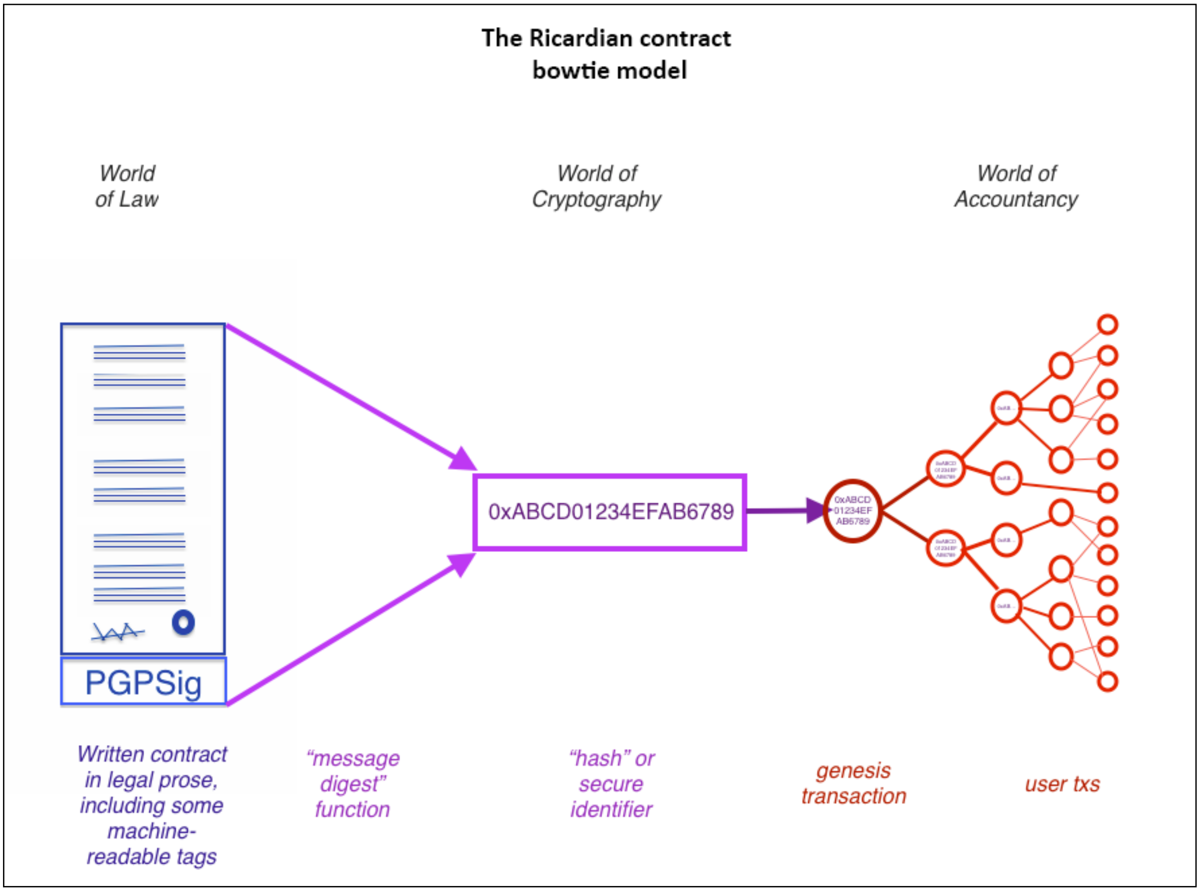
The Ricardian contract bowtie model
World of Law
World of Cryptography
World of Accountancy
PGPSig
0xABCD01234EFAB6789
0xABCD 01234EF AB6789
Written contract in legal prose, including some machine-readable tags
"message digest" function
"hash" or secure identifier
genesis transaction
user txs

Figure 10.1: The Ricardian Contract bowtie diagram

A Ricardian contract is different from a smart contract in the sense that a smart contract does not include any contractual document and is focused purely on the execution of the contract. A Ricardian contract, on the other hand, is more concerned with the semantic richness and production of a document that contains contractual legal prose. The semantics of a contract can be divided into two types: operational semantics and denotational semantics.

The first type defines the actual execution, correctness, and safety of the contract, and the latter is concerned with the real-world meaning of the full contract. Some researchers have differentiated between smart contract code and smart legal contracts, where a smart contract is only concerned with the execution of the contract. The second type encompasses both the denotational and operational semantics of a legal agreement. It perhaps makes sense to categorize smart contracts based on the difference between the semantics, but it is better to consider a smart contract as a standalone entity that is capable of encoding legal prose and code (business logic).

In Bitcoin, a straightforward implementation of basic smart contracts (conditional logic) can be observed, which is entirely oriented toward the execution and performance of the contract, whereas a Ricardian contract is more geared toward producing a document that is understood by humans with some parts that a computer program can understand. This can be viewed as legal semantics versus operational performance (semantics versus performance), as shown in the following diagram. The diagram shows that Ricardian contracts are more semantically-rich, whereas smart contracts are more performance-rich. This concept was initially proposed by *Ian Grigg* in his paper, *On the intersection of Ricardian and Smart Contracts*.

> The paper is available at `https://iang.org/papers/intersection_ricardian_smart.html`.

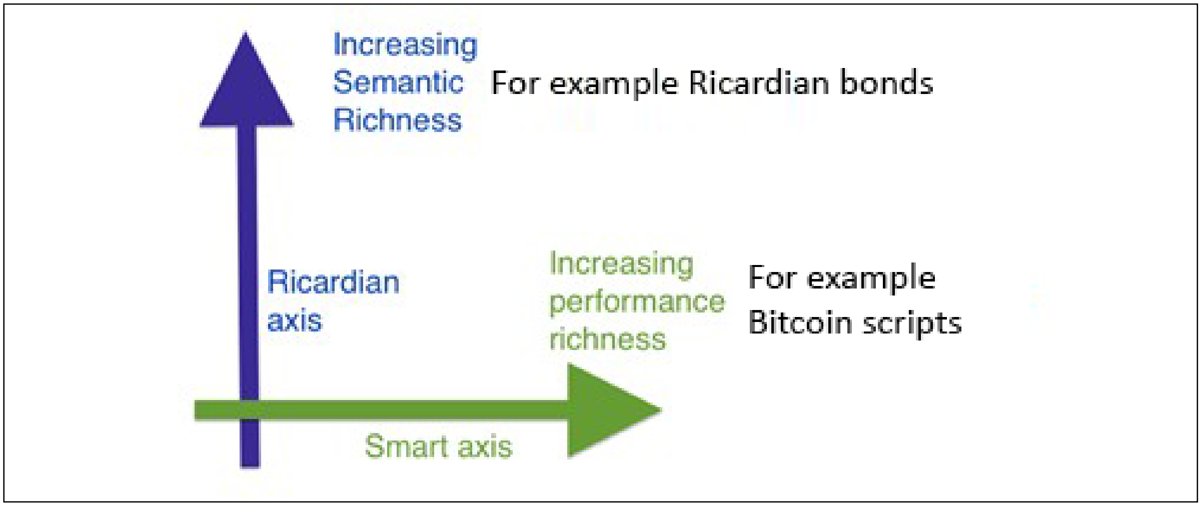

Figure 10.2: Diagram explaining that performance versus semantics is an orthogonal issue, as described by Ian Grigg; it is slightly modified to show examples of different types of contracts on both axes

A smart contract is made up of both of these elements (performance and semantics) embedded together, which completes the ideal model of a smart contract.

A Ricardian contract can be represented as a tuple of three objects, namely **prose**, **parameters**, and **code**. Prose represents the legal contract in natural language; code represents the program that is a computer-understandable representation of legal prose; and the parameters join the appropriate parts of the legal contract to the equivalent code.

Ricardian contracts have been implemented in many systems, such as CommonAccord (`http://www.commonaccord.org`) and OpenBazaar (`https://openbazaar.org`).

Now that we understand what Ricardian contracts are, let's take a look at the concept of **smart contract templates**. These have been built on the idea of Ricardian contracts, and they aim to support the management of the entire life cycle of legal smart contracts.

Smart contract templates

Smart contracts can be implemented in any industry where they are required, but the most popular use cases relate to the financial sector. This is because blockchain first found many use cases in the finance industry and, therefore, sparked enormous research interest in the financial industry long before other areas. Recent work in the smart contract space specific to the financial sector has proposed the idea of smart contract templates. The idea is to build standard templates that provide a framework to support legal agreements for financial instruments.

Christopher D. Clack et al. proposed this idea in their paper published in 2016, named *Smart Contract Templates: Foundations, design landscape and research directions*.

The paper is available at `https://arxiv.org/pdf/1608.00771.pdf`.

The paper also suggested that **domain-specific languages** (**DSLs**) should be built to support the design and implementation of smart contract templates. A language named **common language for augmented contract knowledge** (**CLACK**) has been proposed, and research has started to develop this language. This language is intended to be very rich and is expected to provide a large variety of functions ranging from supporting legal prose to the ability to be executed on multiple platforms and cryptographic functions.

Clack et al. also carried out work to develop smart contract templates that support legally-enforceable smart contracts. This proposal has been discussed in their research paper, *Smart Contract Templates: essential requirements and design options*.

This paper is available at `https://arxiv.org/pdf/1612.04496.pdf`.

The main aim of this paper is to investigate how legal prose could be linked with code using a markup language. It also covers how smart legal agreements can be created, formatted, executed, and serialized for storage and transmission. This work is ongoing and remains an open area for further research and development.

Contracts in the finance industry are not a new concept, and various DSLs are already in use in the financial services industry to provide a specific language for a particular domain. For example, there are DSLs available that support the development of insurance products, represent energy derivatives, or are being used to build trading strategies.

A comprehensive list of financial DSLs can be found at `http://www.dslfin.org/resources.html`.

It is also essential to understand the concept of DSLs, as this type of programming language can be developed to program smart contracts. DSLs are different from **general-purpose programming languages** (**GPLs**). DSLs have limited expressiveness for a particular application or area of interest. These languages possess a small set of features that are sufficient and optimized for a specific domain only. Unlike GPLs, they are not suitable for building large general-purpose application programs.

Based on the design philosophy of DSLs, it can be envisaged that such languages will be developed specifically to write smart contracts. Some work has already been done, and **Solidity** is one such language that has been introduced with the Ethereum blockchain to write smart contracts. Vyper is another language that has been recently introduced for Ethereum smart contract development.

This idea of DSLs for smart contract programming can be further extended to a **GPL**. A smart contract modeling platform can be developed where a domain expert (not a programmer but a front desk dealer, for example) can use a graphical user interface and a canvas (drawing area) to define and illustrate the definition and execution of a financial contract. Once the flow is drawn and completed, it can be emulated first to test it and then be deployed from the same system to the target platform, which can be a smart contract on a blockchain or even a complete **decentralized application** (**DApp**). This concept is also not new, and a similar approach is already used in a non-blockchain domain, in the Tibco StreamBase product, which is a Java-based system used for building event-driven, high-frequency trading systems.

It has been proposed that research should also be conducted in the area of developing high-level DSLs that can be used to program a smart contract in a user-friendly graphical user interface, thus allowing a non-programmer domain expert (for example, a lawyer) to design smart contracts.

Apart from DSLs, there is also a growing interest in using general-purpose, already established programming languages like Java, Go, and C++ to be used for smart contract programming. This idea is appealing, especially from a usability point of view, where a programmer who is already familiar with, for example, Java, can use their skills to write Java code instead of learning a new language. The high-level language code can then be compiled into a low-level bytecode for execution on the target platform. There are already some examples of such systems, such as in EOSIO blockchains, where C++ can be used to write smart contracts, which are compiled down to the web assembly for execution.

An inherent limitation with smart contracts is that they are unable to access any external data. The concept of oracles was introduced to address this issue. An oracle is an off-chain source of information that provides the required information to the smart contracts on the blockchain.

> Some earlier discussions and thoughts about oracles can be found here:
>
> `https://bitcointalk.org/index.php?topic=8821.msg133523#msg133523`
>
> `https://github.com/orisi/wiki/wiki/Orisi-White-Paper`
>
> `http://gavintech.blogspot.com/2014/06/bit-thereum.html`

Now we will discuss oracles in more detail.

Oracles

Oracles are an essential component of the smart contract and blockchain ecosystem. The limitation with smart contracts is that they cannot access external data because blockchains are closed systems without any direct access to the real world. This external data might be required to control the execution of some business logic in the smart contract; for example, the stock price of a security product that is required by the contract to release dividend payments. In such situations, oracles can be used to provide external data to smart contracts. An oracle can be defined as an interface that delivers data from an external source to smart contracts. Oracles are trusted entities that use a secure channel to transfer off-chain data to a smart contract.

The following diagram shows a generic model of an oracle and smart contract ecosystem:

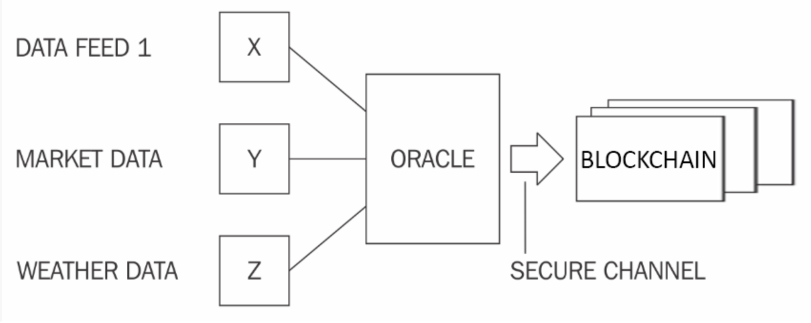

Figure 10.3: A generic model of an oracle and smart contract ecosystem

Depending on the industry and use case requirements, oracles can deliver different types of data ranging from weather reports, real-world news, and corporate actions to data coming from an **Internet of Things** (**IoT**) device.

A list of some of the common use cases of oracles is shown here:

Type of data	**Examples**	**Use case**
Market data	Live price feeds of financial instruments. Exchange rates, performance, pricing, and historic data of commodities, indices, equities, bonds, and currencies.	DApps related to financial services, for example, decentralized exchanges and **decentralized finance** (**DeFi**)
Political events	Election results	Prediction markets
Travel information	Flight schedules and delays	Insurance DApps
Weather information	Flooding, temperature, and rain data	Insurance DApps
Sports	Results of football, cricket, and rugby matches	Prediction markets
Telemetry	Hardware IoT devices, sensor data, vehicle location, and vehicle tracker data	Insurance DApps Vehicle fleet management DApps

here are different methods used by oracles to write data into a blockchain, depending on the type of blockchain used. For example, in a Bitcoin blockchain, an oracle can write data to a specific transaction, and a smart contract can monitor that transaction in the blockchain and read the data. Other methods include storing the fetched data in a smart contract's storage, which can then be accessed by other smart contracts on the blockchain via requests between smart contracts depending on the platform. For example, in Ethereum, this can be achieved by using message calls.

The standard mechanics of how oracles work is presented here:

1. A smart contract sends a request for data to an oracle.
2. The request is executed and the required data is requested from the source. There are various methods of requesting data from the source. These methods usually involve invoking APIs provided by the data provider, calling a web service, reading from a database (for example, in enterprise integration use cases where the required data may exist on a local enterprise legacy system), or requesting data from another blockchain. Sources can be any external off-chain data provider on the internet or in an internal enterprise network.
3. The data is sent to a notary to generate cryptographic proof (usually a digital signature) of the requested data to prove its validity (authenticity). Usually, TLSNotary is used for this purpose (`https://tlsnotary.org`). Other techniques include **Android proofs**, **Ledger proofs**, and **trusted hardware-assisted proofs**, which we will explain shortly.
4. The data with the proof of validity is sent to the oracle.
5. The requested data with its proof of authenticity can be optionally saved on a decentralized storage system such as Swarm or IPFS and can be used by the smart contract/blockchain for verification. This is especially useful when the proofs of authenticity are of a large size and sending them to the requesting smart contracts (storing them on the chain) is not feasible.
6. Finally, the data, with the proof of validity, is sent to the smart contract.

This process can be visualized in the following diagram:

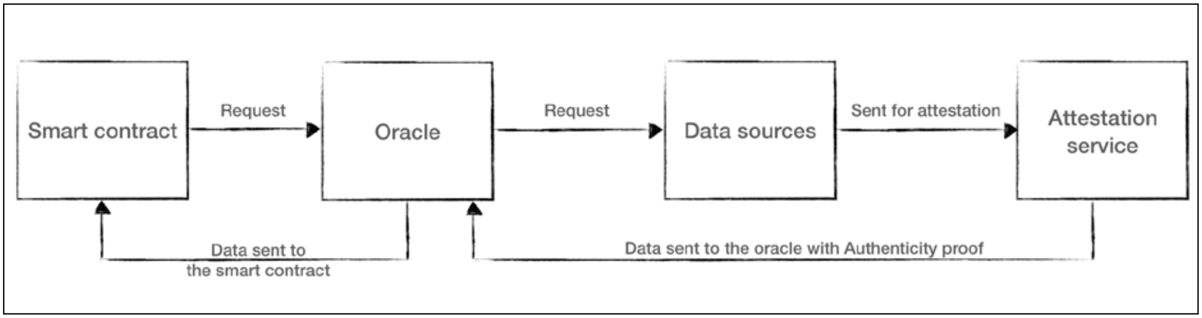

Figure 10.4: A generic oracle data flow

The preceding diagram shows the generic data flow of a data request from a smart contract to the oracle. The oracle then requests the data from the data source, which is then sent to the attestation service for notarization. The data is sent to the oracle with proof of authenticity. Finally, the data is sent to the smart contract with cryptographic proof (authenticity proof) that the data is valid.

Due to security requirements, oracles should also be capable of digitally signing or digitally attesting the data to prove that the data is authentic. This proof is called **proof of validity** or **proof of authenticity**.

Smart contracts subscribe to oracles. Smart contracts can either pull data from oracles, or oracles can push data to smart contracts. It is also necessary that oracles should not be able to manipulate the data they provide and must be able to provide factual data. Even though oracles are trusted (due to the associated proof of authenticity of data), it may still be possible that, in some cases, the data is incorrect due to manipulation or a fault in the system. Therefore, oracles must not be able to modify the data. This validation can be provided by using various cryptographic proofing schemes. We will now introduce different mechanisms to produce cryptographic proof of data authenticity.

Software and network-assisted proofs

As the name suggests, these types of proofs make use of software, network protocols, or a combination of both to provide validity proofs. One of the prime examples of such proofs is TLSNotary, which is a technology developed to be primarily used in the PageSigner project (`https://tlsnotary.org/pagesigner.html`) to provide web page notarization. This mechanism can also be used to provide the required security services to oracles.

TLSNotary

This protocol provides a piece of irrefutable evidence to an auditor that specific web traffic has occurred between a client and a server. It is based on **Transport Layer Security** (**TLS**), which is a standard security mechanism that enables secure, bidirectional communication between hosts. It is extensively used on the internet to secure websites and allow HTTPS traffic.

A discussion of the internals of this protocol is beyond the scope of this book. Interested readers can read the standards document at `http://www.ietf.org/rfc/rfc2246.txt`. The link provided is only for TLS version 1.0 as TLSNotary only supports TLS version 1.0 or 1.1.

The key idea behind using TLSNotary is to utilize the TLS handshake protocol's feature, which allows the splitting of the TLS master key into three parts. Each part is allocated to the server, the auditee, and the auditor. The oracle service provider (`https://provable.xyz`) becomes the auditee, whereas an **Amazon Web Services** (**AWS**) instance, which is secure and locked down, serves as the auditor.

In summary, to prove the authenticity of the data retrieved by oracles from external sources, attestation mechanisms such as TLSNotary are used, which produce verifiable and auditable proofs of communication between the data source and the oracle. This proof of authenticity ensures that the data fed back to the smart contract is indeed retrieved from the source.

TLS-N based mechanism

This mechanism is one of the latest developments in this space. TLS-N is a TLS extension that provides secure non-repudiation guarantees. This protocol allows you to create privacy-preserving and non-interactive proofs of the content of a TLS session. TLS-N based oracles do not need to trust any third-party hardware such as Intel SGX or TLSNotary type service to provide authenticity proofs of data web content (data) to the blockchain. In contrast to TLSNotary, this scheme works on the latest TLS 1.3 standard, which allows for improved security. More information regarding this protocol is available at `https://eprint.iacr.org/2017/578.pdf`.

Hardware device-assisted proofs

As the name suggests, these proofs rely on some hardware elements to provide proof of authenticity. In other words, they require specific hardware to work. Different mechanisms come under this category, and we will briefly introduce those next.

Android proof

This proof relies on Android's SafetyNet software attestation and hardware attestation to create a provably secure and auditable device. SafetyNet validates that a genuine Android application is being executed on a secure, safe, and untampered hardware device. Hardware attestation validates that the device has the latest version of the OS, which helps to prevent any exploits that existed due to vulnerabilities in the previous versions of the OS. This secure device is then used to fetch data from third-party sources, ensuring tamper-proof HTTPS connections. The very use of a provably secure device provides the guarantee and confidence (that is, a proof of authenticity) that the data is authentic.

> More information regarding SafetyNet and hardware attestation is available here:
>
> `https://developer.android.com/training/safetynet`
>
> `https://developer.android.com/training/articles/security-key-attestation.html`

Ledger proof

Ledger proof relies on the hardware cryptocurrency wallets built by the ledger company (`https://www.ledger.com`). Two hardware wallets, **Ledger Nano S** and **Ledger Blue**, can be used for these proofs. The primary purpose of these devices is as secure hardware cryptocurrency wallets. However, due to the security and flexibility provided by these devices, they also allow developers to build other applications for this hardware. These devices run a particular OS called **Blockchain Open Ledger Operating System** (**BOLOS**), which, via several kernel-level APIs, allows device and code attestation to provide a provably secure environment.

The secure environment provided by the device can also prove that the applications that may have been developed by oracle service providers and are running on the device are valid, authentic, and are indeed executing on the **Trusted Execution Environment** (**TEE**) of the ledger device. This environment, supported by both code and device attestation, provides an environment that allows you to run a third-party application in a secure and verifiable ledger environment to provide proof of data authenticity. Currently, this service is used by Provable, an oracle service, to provide untampered random numbers to smart contracts.

> Ledger proofs, Android proofs, and TLSNotary proofs are used in provable oracles. The official documentation for these methods can be found here: `http://docs.provable.xyz`.

Currently, as these devices do not connect to the internet directly, the ledger devices cannot be used to fetch data from the internet.

Trusted hardware-assisted proofs

This type of proof makes use of trusted hardware, such as TEEs. A prime example of such a hardware device is Intel SGX. A general approach that is used in this scenario is to rely on the security guarantees of a secure and trusted execution provided by the secure element or enclave of the TEE device.

A prime example of a trusted hardware-assisted proof is Town Crier (`https://www.town-crier.org`), which provides an authenticated data feed for smart contracts. It uses Intel SGX to provide a security guarantee that the requested data has come from an existing trustworthy resource.

> Intel SGX technology is developed by Intel, which provides a hardware TEE. More information about this is available here: `https://software.intel.com/en-us/sgx`. It is used by many different oracle service providers such as Town Crier and iExec (`https://iex.ec/decentralized-oracles/`).

Town Crier also provides a confidentiality service, which allows you to run confidential queries. The query for the data request is processed inside SGX Enclave, which provides a trusted execution guarantee, and the requested data is transmitted using a TLS-secured network connection, which provides additional data integrity guarantees.

> It should be noted that in all of the proof techniques mentioned earlier, there are many types of resources used to provide security guarantees, including hardware, network, and software. The categorization provided here is based on the principal element, either hardware or software, which plays a critical role in the overall security mechanism to provide security.

An issue can already be seen here, and that is the issue of trust. With oracles, we are effectively trusting a third party to provide us with the correct data. What if these data sources turn malicious, or simply due to a fault start provide incorrect data to the oracles? What if the oracle itself fails or the data source stops sending data? This issue can then damage the whole blockchain trust model. This phenomenon is called the **Blockchain oracle problem**. How do you trust a third party about the quality and authenticity of the data they provide? This question is especially real in the financial world, where market data must be accurate and reliable.

There are several proposed ways to overcome this issue. These solutions range from merely trusting a reputable third party to decentralized oracles. We have discussed some of the attestation techniques earlier; however, a third party due to a genuine fault or a malicious intent may still provide data that is incorrect. Even if it is attested later on, the actual data itself is not guaranteed to be accurate. It might be acceptable for a smart contract designer in a use case to accept data for an oracle that is provided by a large, reputable, and trusted third party. For example, the source of the data can be from a reputable weather reporting agency or airport information system directly relaying the flight delays, which can give some level of confidence. However, the issue of centralization remains.

> The blockchain oracle problem can be defined formally as the conflict of trust between presumably trusted Oracles (trusted third-party data sources) and completely trustless blockchain.

Based on the evolution of blockchain over the last few years, several types of blockchain oracles have emerged. We informally provided some background on these earlier, but now we will define these formally.

Types of blockchain oracles

There are various types of blockchain oracles, ranging from simple software oracles to complex hardware assisted and decentralized oracles. Broadly speaking, we can categorize oracles into two categories: inbound oracles and outbound oracles. The following section will examine some of these in more detail.

Inbound oracles

This class represents oracles that receive incoming data from external services, and feed it into the smart contract. We will shortly discuss software, hardware, and several other types of inbound oracle.

Software oracles

These oracles are responsible for acquiring information from online services on the Internet. This type of oracle is usually used to source data such as weather information, financial data (stock prices, for example), travel information and other types of data from third-party providers. The data source can also be an internal enterprise system, which may provide some enterprise-specific data. These types of oracle can also be called standard or simple oracles.

Hardware oracles

This type of oracle is used to source data from hardware sources such as IoT devices or sensors. This is useful in use cases such as insurance-related smart contracts where telemetry sensors provide certain information, for example, vehicle speed and location. This information can be fed into the smart contract dealing with insurance claims and payouts to decide whether to accept a claim or not. Based on the information received from the source hardware sensors, the smart contract can decide whether to accept or reject the claim.

These oracles are useful in any situation where real-world data from physical devices is required. However, this approach requires a mechanism in which hardware devices are tamper-proof or tamper-resistant. This level of security can be achieved by providing cryptographic evidence (non-repudiation and integrity) of IoT device's data and an anti-tampering mechanism on the IoT device, which renders the device useless in case of tampering attempts.

Computation oracles

These oracles allow computing-intensive calculations to be performed off-chain. As blockchain is not suitable for performing compute-intensive operations, a blockchain (that is, a smart contract on a blockchain) can request computations to be performed on off-chain high-performance computing infrastructure and get the verified results back via an oracle. The use of oracle, in this case, provides data integrity and authenticity guarantees.

An example of such an oracle is Truebit (`https://truebit.io`). It allows a smart contract to submit computation tasks to oracles, which are eventually completed by miners in return for an incentive.

Aggregation based oracles

In this scenario, a single value is sourced from many different feeds. As an example, this single value can be the price of a financial instrument, and it can be risky to rely upon only one feed. To mitigate this problem, multiple data providers can be used where all of these feeds are inspected, and finally, the price value that is reported by most of the feeds can be picked up. The assumption here is that if the majority of the sources reports the same price value, then it is likely to be correct. The collation mechanism depends on the use case: sometimes it's merely an average of multiple values, sometimes a median is taken of all the values, and sometimes it is the maximum value. Regardless of the aggregation mechanism, the essential requirement here is to get the value that is valid and authentic, which eventually feeds into the system.

An excellent example of price feed oracles is MakerDAO (`https://makerdao.com/en/`) price feed oracle (`https://developer.makerdao.com/feeds/`), which collates price data from multiple external price feed sources and provides a median ETHUSD price to MakerDAO.

Crowd wisdom driven oracles

This is another way that the blockchain oracle problem can be addressed where a single source is not trusted. Instead, multiple public sources are used to deduce the most appropriate data eventually. In other words, it solves the problem where a single source of data may not be trustworthy or accurate as expected. If there is only one source of data, it can be unreliable and risky to rely on entirely. It may turn malicious or become genuinely faulty.

In this case, to ensure the credibility of data provided by third-party sources for oracles, the data is sourced from multiple sources. These sources can be users of the system or even members of the general public who have access to and have knowledge of some data, for example, a political event or a sporting event where members of the public know the results and can provide the required data. Similarly, this data can be sourced from multiple different news websites. This data can then be aggregated, and if a sufficiently high number of the same information is received from multiple sources, then there is an increased likelihood that the data is correct and can be trusted.

Decentralized oracles

Another type of oracles, which primarily emerged due to the decentralization requirements, is called **decentralized** oracles. Remember that in all types of oracles discussed so far, there are some trust requirements to be placed in a trusted third party. As blockchain platforms such as Bitcoin and Ethereum are fully decentralized, it is expected that oracle services should also be decentralized. This way, we can address the *Blockchain Oracle Problem*.

This type of oracle can be built based on a distributed mechanism. It can also be envisaged that the oracles can find themselves source data from another blockchain, which is driven by distributed consensus, thus ensuring the authenticity of data. For example, one institution running their private blockchain can publish their data feed via an oracle that can then be consumed by other blockchains.

A decentralized oracle essentially allows off-chain information to be transferred to a blockchain without relying on a trusted third party.

Augur (visit `https://www.augur.net/whitepaper.pdf` for Jack Peterson et al.'s essay, *A Decentralized Oracle and Prediction Market Platform*) is a prime example of such type of oracles. The Augur white paper is also available here: `https://arxiv.org/abs/1501.01042`.

The core idea behind Augur's oracle is that of crowd wisdom-supported oracles, in which the information about an event is acquired from multiple sources and aggregated into the most likely outcome. The sources in case of Augur are financially motivated reporters who are rewarded for correct reporting and penalized for incorrect reporting.

> Decentralized, crowd wisdom based and aggregation supported oracles can be categorized into a broader category of oracles called "consensus driven oracles". Augur is based on Crowd Wisdom based oracle.

Smart oracles

An idea of smart oracle has also been proposed by **Ripple labs** (**codius**). Its original whitepaper is available at `https://github.com/codius/codius-wiki/wiki/White-Paper#from-oracles-to-smart-oracles`. Smart oracles are entities just like oracles, but with the added capability of executing contract code. Smart oracles proposed by Codius run using Google Native Client, which is a sandboxed environment for running untrusted x86 native code.

Outbound oracles

This type, also called **reverse oracles**, are used to send data out from the blockchain smart contracts to the outside world. There are two possible scenarios here; one is where the source blockchain is a producer of some data such as blockchain metrics, which are needed for some other blockchain. The actual data somehow needs to be sent out to another blockchain smart contract. The other scenario is that an external hardware device needs to perform some physical activity in response to a transaction on-chain. However, note that this type of scenario does not necessarily need an oracle, because the external hardware device can be sent a signal as a result of the smart contract event.

On the other hand, it can be argued that if the hardware device is running on an external blockchain, then to get data from the source chain to the target chain, undoubtedly, will need some security guarantees that oracle infrastructure can provide. Another situation is where we need to integrate legacy enterprise systems with the blockchain. In that case, the outbound oracle would be able to provide blockchain data to the existing legacy systems. An example scenario is the settlement of a trade done on a blockchain that needs to be reported to the legacy settlement and backend reporting systems.

Now that we have discussed different types of oracles, we now introduce different service providers that provide these services. Several service providers provide oracle services for blockchain, some of these we introduce following.

Blockchain oracle services

Various online services are now available that provide oracle services. These can also be called **oracle-as-a-service platforms**. All of these services aim to enable a smart contract to securely acquire the off-chain data it needs to execute and make decisions:

- Town Crier: `https://www.town-crier.org`
- Provable: `https://provable.xyz`
- Witnet: `https://witnet.io`
- Chainlink: `https://chain.link`
- The Realitio project: `https://realit.io`
- TrueBit: `https://truebit.io`
- iExec: `https://iex.ec`

Another service at `https://smartcontract.com/` is also available, where Ethereum, Bitcoin, and Town Crier oracles can be created. It allows smart contracts to connect to applications and allows you to feed data into the smart contracts from off-chain sources.

There are many oracle services available now, and it is challenging to cover all of them here. A random selection is presented in the preceding list.

Now that we've covered oracles in detail, let's take a look at smart contracts again and see how smart contracts can be deployed at a fundamental level.

Deploying smart contracts

Smart contracts may or may not be deployed on a blockchain, but it makes sense to do so on a blockchain due to the security and decentralized consensus mechanism provided by the blockchain. Ethereum is an example of a blockchain platform that natively supports the development and deployment of smart contracts. We will cover Ethereum in more detail later in this book, in *Chapter 11*, *Ethereum 101*. Smart contracts on an Ethereum blockchain are typically part of a broader DApp.

In comparison, in a Bitcoin blockchain, the transaction timelocks, such as the `nLocktime` field, the `CHECKLOCKTIMEVERIFY` (**CLTV**), and the `CHECKSEQUENCEVERIFY` script operator in the Bitcoin transaction, can be seen as an enabler of a simple version of a smart contract. These timelocks enable a transaction to be locked until a specified time or until a number of blocks, thus enforcing a basic contract that a certain transaction can only be unlocked if certain conditions (elapsed time or number of blocks) are met. For example, you can implement conditions such as *Pay party X, N number of bitcoins after 3 months*. However, this is very limited and should only be viewed as an example of a basic smart contract. In addition to the example mentioned earlier, Bitcoin scripting language, though limited, can be used to construct basic smart contracts. One example of a basic smart contract is to fund a Bitcoin address that can be spent by anyone who demonstrates a **hash collision attack**.

This was a contest that was announced on the Bitcointalk forum where bitcoins were set as a reward for whoever manages to find hash collisions (we discussed this concept in *Chapter 3*, *Symmetric Cryptography*) for hash functions. This conditional unlocking of Bitcoin solely on the demonstration of a successful attack is a basic type of smart contract.

This idea was presented on the Bitcointalk forum, and more information can be found at `https://bitcointalk.org/index.php?topic=293382.0`. This can be considered a basic form of a smart contract.

Various other blockchain platforms support smart contracts such as Monax, Lisk, Counterparty, Stellar, Hyperledger Fabric, Axoni core, Neo, EOSIO, and Tezos. Smart contracts can be developed in various languages, either DSLs or general-purpose languages. The critical requirement, however, is determinism, which is very important because it is vital that regardless of where the smart contract code executes, it produces the same result every time and everywhere. This requirement of the deterministic nature of smart contracts also implies that smart contract code is absolutely bug-free. Validation and verification of smart contracts is an active area of research and a detailed discussion of this topic will be presented in *Chapter 21*, *Scalability and Other Challenges*.

Various languages have been developed to build smart contracts such as Solidity, which runs on **Ethereum Virtual Machine** (**EVM**). It's worth noting that there are platforms that already support mainstream languages for smart contract development, such as Lisk, which supports JavaScript. Another prominent example is Hyperledger Fabric, which supports Golang, Java, and JavaScript for smart contract development. A more recent example is EOSIO, which supports writing smart contracts in C++.

Security is of paramount importance for smart contracts. However, there are many vulnerabilities discovered in prevalent blockchain platforms and relevant smart contract development languages. These vulnerabilities result in some high-profile incidents, such as the DAO attack.

The DAO

The **Decentralized Autonomous Organization** (**DAO**), started in April 2016, was a smart contract written to provide a platform for investment. Due to a bug, called the **reentrancy bug**, in the code, it was hacked in June 2016. An equivalent of approximately 3.6 million ether (roughly 50 million US dollars) was siphoned out of the DAO into another account.

Even though the term hacked is used here, it was not really hacked. The smart contract did what it was asked to do but due to the vulnerabilities in the smart contracts, the attacker was able to exploit it. It can be seen as an unintentional behavior (a bug) that programmers of the DAO did not foresee. This incident resulted in a hard fork on the Ethereum blockchain, which was introduced to recover from the attack.

The DAO attack exploited a vulnerability (reentrancy bug) in the DAO code where it was possible to withdraw tokens from the DAO smart contract repeatedly before giving the DAO contract a chance to update its internal state to indicate that how many DAO tokens have been withdrawn. The attacker was able to withdraw DAOs. However, before the smart contract could update its state, the attacker withdrew the tokens again. This process was repeated many times, but eventually, only a single withdrawal was logged by the smart contract, and the contract also lost record of any repeated withdrawals.

The notion of *code is the law or unstoppable smart contracts* should be viewed with some skepticism as the implementation of these concepts is still not mature enough to deserve complete and unquestionable trust. This is evident from the events after the DAO incident, where the Ethereum foundation was able to stop and change the execution of the DAO by introducing a hard fork on the Ethereum blockchain. Though this hard fork was introduced for genuine reasons, it goes against the true spirit of decentralization, immutability, and the notion that *code is the law*. Subsequently, resistance against this hard fork resulted in the creation of Ethereum Classic, where a large number of users decided to keep mining on the old chain. This chain is the original, non-forked Ethereum blockchain that still contains the DAO. It can be said that on this chain, *the code is still the law*.

There are some interesting message threads and announcements related to this event, which readers may find informative and entertaining:

- An open letter from *The Attacker* of the DAO: `https://pastebin.com/CcGUBgDG`
- Announcement from Ethereum core dev: `https://twitter.com/avsa/status/745313647514226688`
- Reddit discussion on the hard fork, where points made in favor and against can be seen: `https://www.reddit.com/r/ethereum/comments/4sgihm/ethcore_blog_post_in_support_of_a_hard_fork/`
- Hard fork specification: `https://blog.slock.it/hard-fork-specification-24b889e70703`

The DAO attack highlights the dangers of not formally and thoroughly testing smart contracts. It also highlights the absolute need to develop a formal language for the development and verification of smart contracts. The attack also highlighted the importance of thorough testing to avoid the issues that the DAO experienced. There have been various vulnerabilities discovered in Ethereum over the last few years regarding the smart contract development language. Therefore, it is of utmost importance that a standard framework is developed to address all these issues.

Some work has already begun, for example, an online service at `https://securify.ch`, which provides tools to formally verify smart contract code.

Another example is Michelson (`https://www.michelson.org`) for writing smart contracts in the Tezos blockchain. It is a functional programming language suitable for formal verification. We will introduce the Tezos blockchain in more detail in this book's bonus content pages, here `https://static.packt-cdn.com/downloads/Altcoins_Ethereum_Projects_and_More_Bonus_Content.pdf.`

Vyper is another language that aims to provide a secure language for developing smart contracts for EVM. It aims for goals such as security, simplicity, and audibility. It is a strongly typed language with support for overflow checking and signed integers. All these features make Vyper a reasonable choice for writing secure smart contracts.

Even though many different initiatives are aiming to explore and address the security of smart contracts, this field still requires further research to address limitations in smart contract programming languages.

The security of smart contracts is an area of deep interest for researchers. A major area of interest is the formal verification of smart contracts. We will discuss these topics further in *Chapter 21*, *Scalability and Other Challenges*.

Summary

This chapter began by introducing a history of smart contracts followed by a detailed discussion of the definition of a smart contract. As there is no agreement on the standard definition of a smart contract, we attempted to introduce a definition that encompasses the crux of smart contracts.

An introduction to Ricardian contracts was also provided, and the difference between Ricardian contracts and smart contracts was explained, highlighting the fact that Ricardian contracts are concerned with the definition of the contract, whereas smart contracts are geared toward the actual execution of the contract.

The concept of smart contract templates was also discussed, in which high-quality active research is currently being conducted in academia and industry. Some ideas about the possibility of creating high-level DSLs were also discussed to create smart contracts or smart contract templates. In the later sections of the chapter, oracles were introduced followed by a brief discussion on the DAO along with security issues in DAO and smart contracts.

A discussion regarding formal verification and the security of smart contracts will be presented later in this book, in *Chapter 21*, *Scalability and Other Challenges*.

In the next chapter, we will introduce Ethereum, which is one of the most popular blockchain platforms that inherently supports smart contracts.