Giuseppe Destefanis

giuseppe.Destefanis@brunel.ac.uk

Analysis and auditing tools for Smart Contracts: Large Language Models and Code Interpretation

The AI “buzzword”

We're living in years where artificial intelligence is becoming popular in various fields - from healthcare to finance, software engineering and blockchain technology.

The Rise of Large Language Models

One significant development in AI has been the emergence of Large Language Models.

They can understand and generate human-like text, making them a powerful tool in many areas.

LLMs in Blockchain development

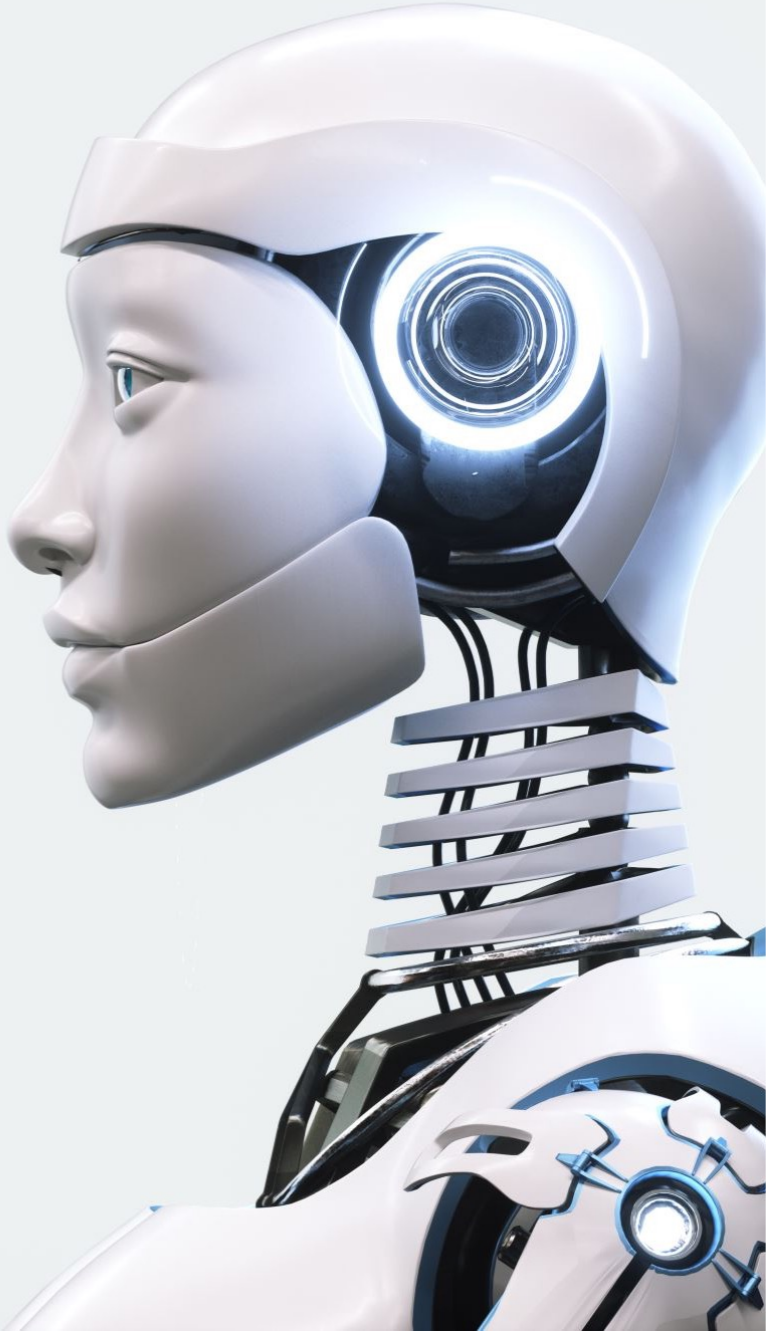
In the blockchain field, LLMs can change how we analyze and test smart contracts.

We will explore how ChatGPT's code interpretation capabilities can advance these critical areas.

AI Evolution

From simple rule-based systems to sophisticated machine learning and deep learning models, artificial intelligence has undergone significant evolution.

Today, we stand at the threshold of a new era defined by Large Language Models.



Large Language Model

Large Language Models are AI models trained on vast amounts of text data.

They are designed to generate human-like text based on the input provided to them.

How LLMs Work

LLMs use patterns in the data they were trained on to generate coherent and contextually appropriate responses.

Models like GPT-3 consist of 175 billion parameters that help them understand and generate complex text.

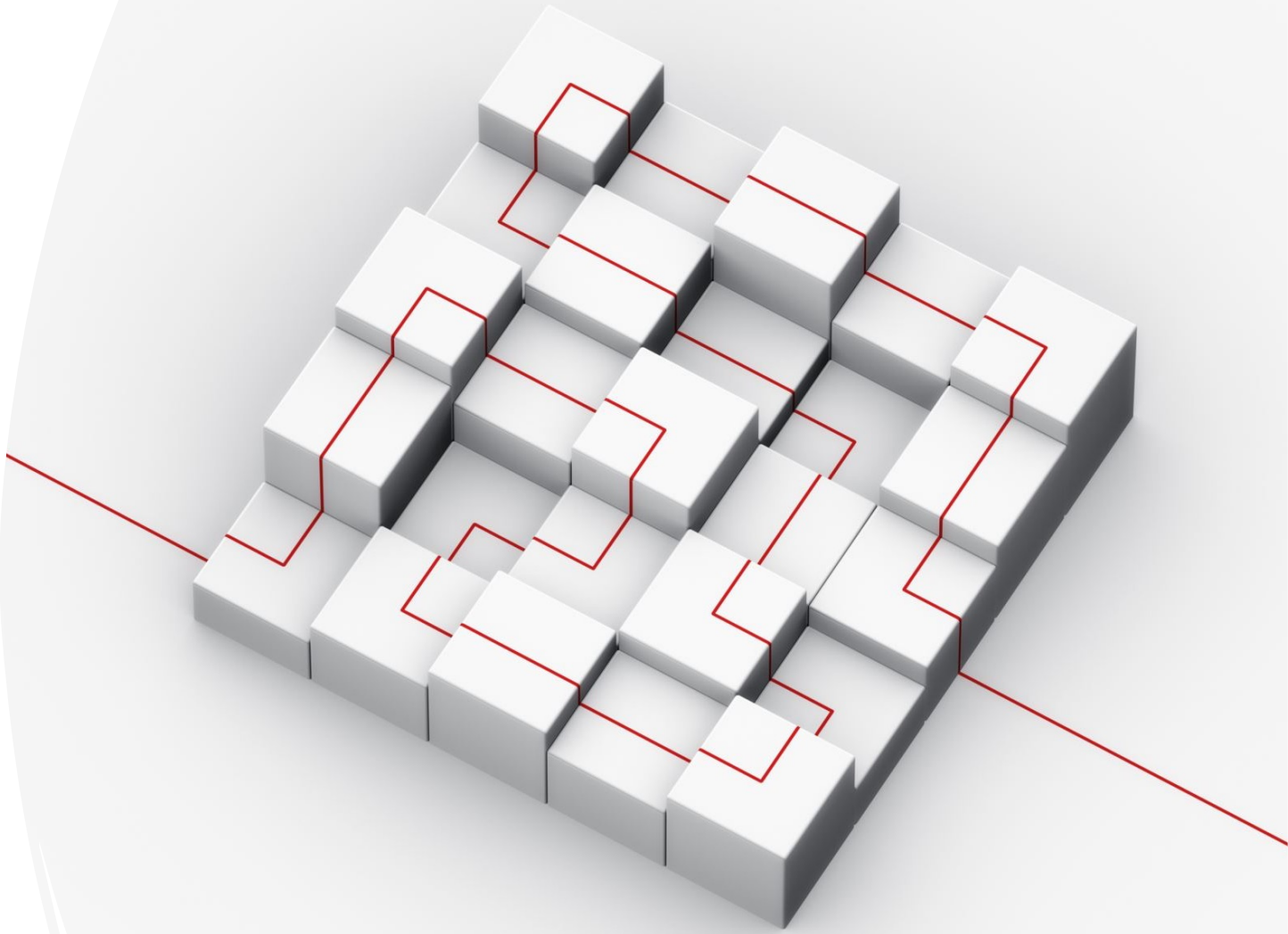
Applications of LLMs

LLMs have broad applications, ranging from drafting emails, creating written content, automating customer service, aiding in research, to more technical applications like code generation and interpretation.

ChatGPT

ChatGPT is a state-of-the-art AI model developed by OpenAI, trained on a diverse range of internet text.

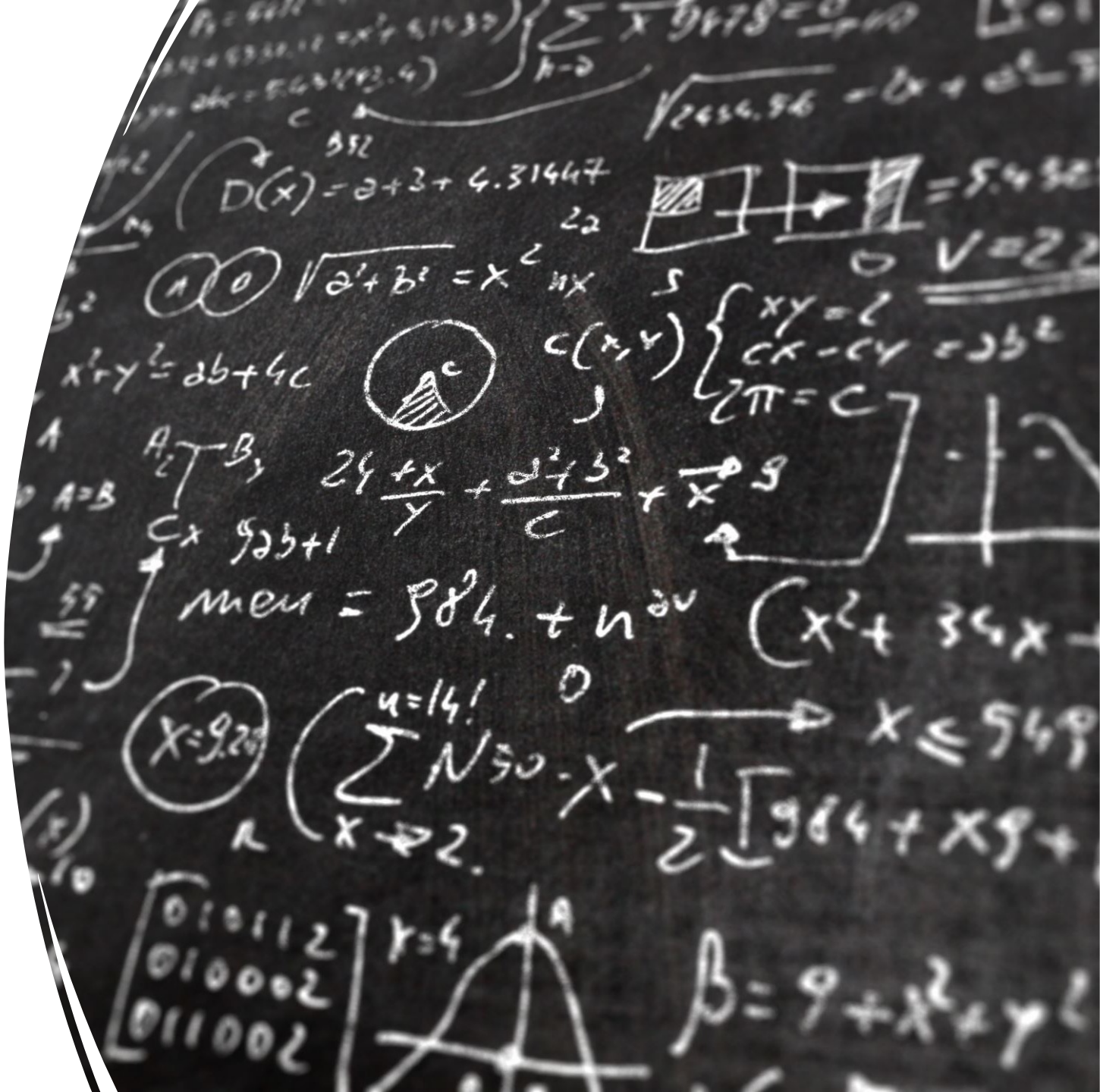
Transformers & GPT: ChatGPT is based on a transformer, which is a type of model architecture used in natural language processing. It is part of the Generative Pretrained Transformer (GPT) series, with GPT-4 being the latest version.



How ChatGPT Works

ChatGPT, like other LLMs, generates predictions by analyzing patterns in the data it was trained on.

It does not know facts, but it uses its training data to generate relevant and coherent outputs.



Language Understanding

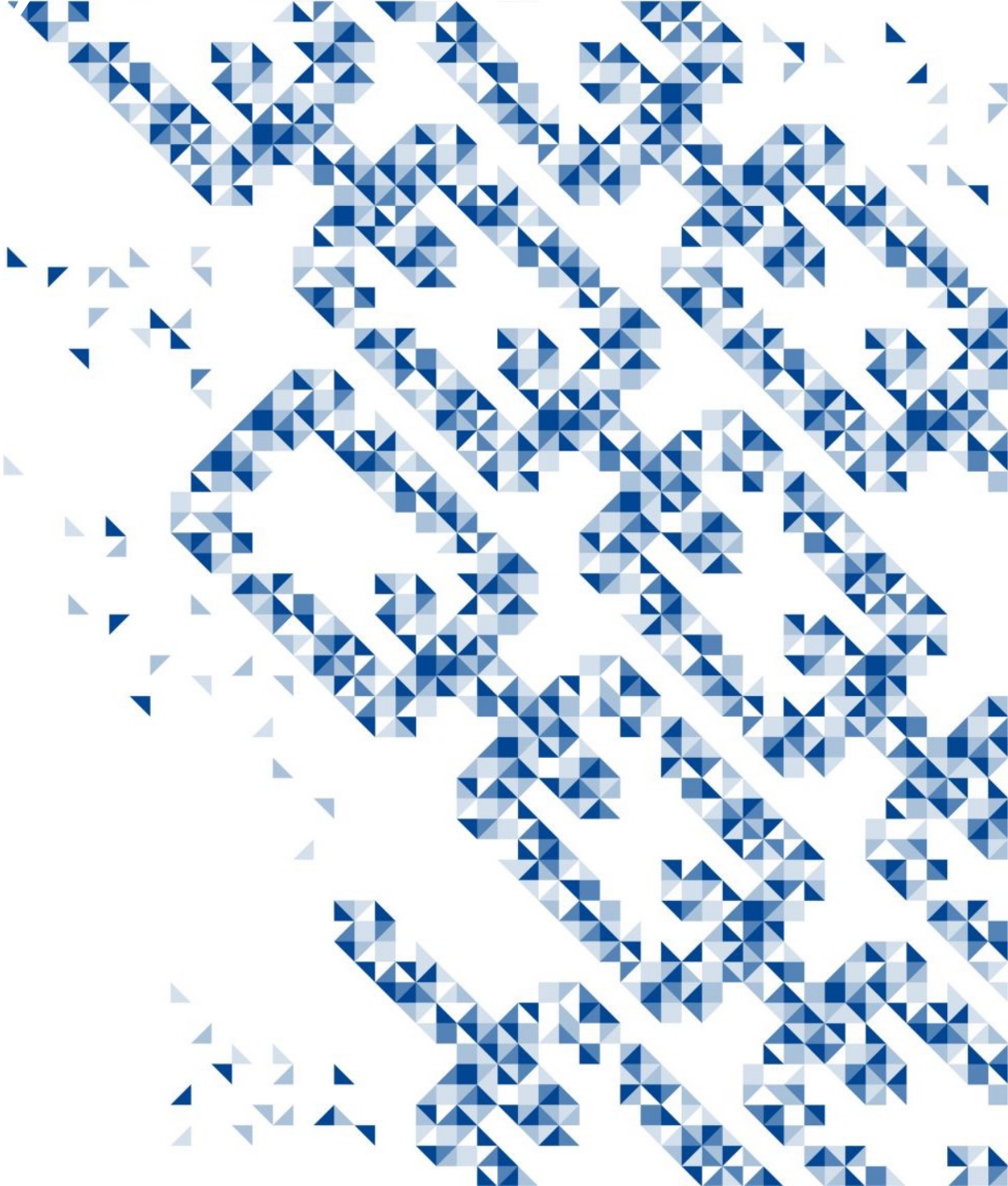
By analyzing the context of the input provided, ChatGPT can generate contextually appropriate responses.

This ability to understand and follow the context makes it seem as if the model "understands" language.

Generating Human-like Text

ChatGPT generates human-like text by predicting what comes next in a piece of text.

It's like a sophisticated autocomplete: it predicts the next word, then the next, and so on, generating sentences and paragraphs.



Code Interpretation

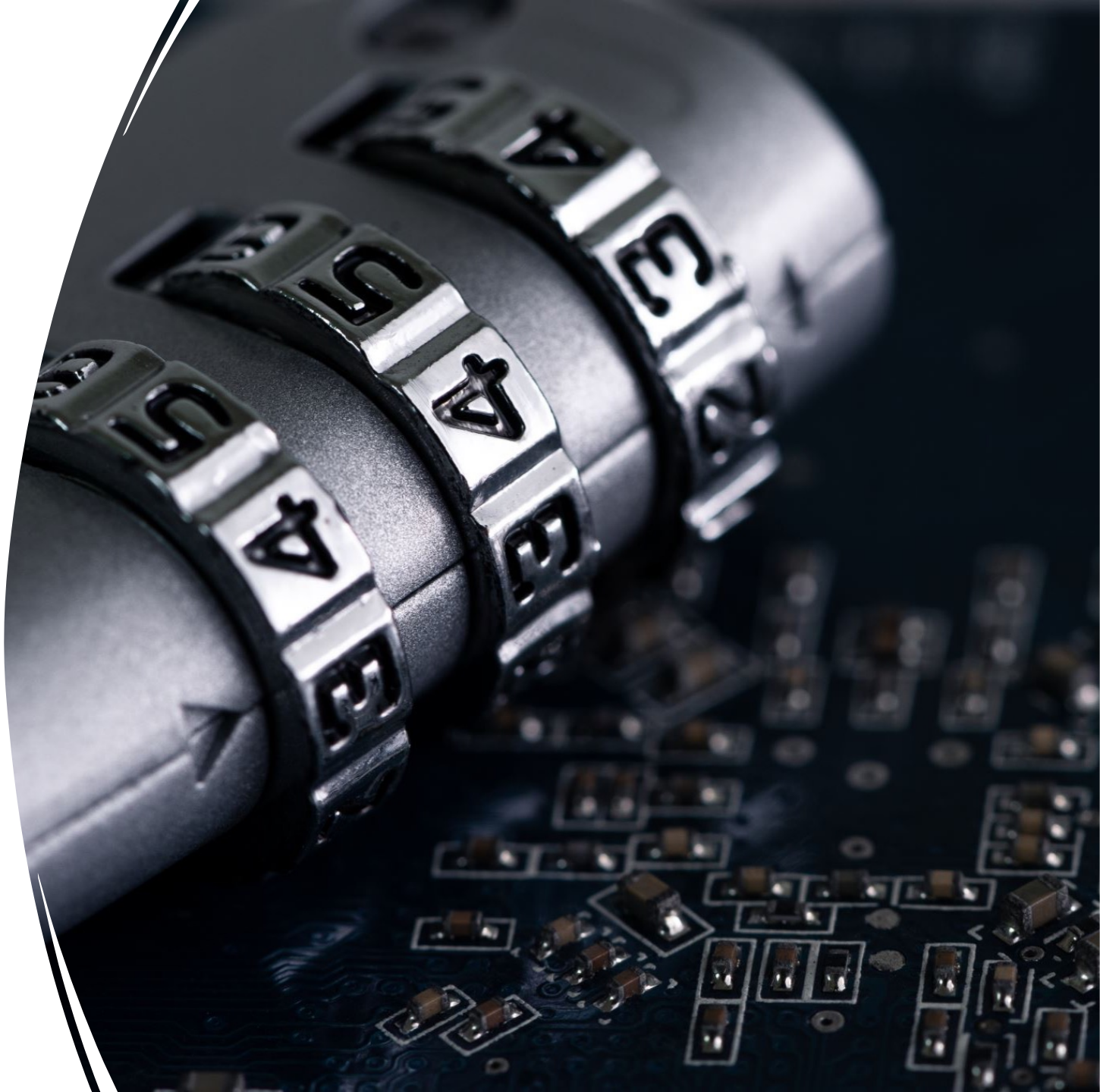
ChatGPT can be used to understand code, making it a potentially valuable tool in software engineering.

It can help read, understand, and explain the functionality of existing code segments.

Code Generation

With sufficient context, ChatGPT can also generate code. This ability opens up exciting possibilities in automating certain coding tasks and assisting programmers.

The ability of ChatGPT to interpret and generate code could revolutionize how we approach software replication studies and smart contract analysis.



Smart Contracts vulnerabilities

A smart contract vulnerability refers to a flaw or weakness in a smart contract's code that could be exploited to perform unauthorized actions, manipulate the contract's intended behavior, or cause it to behave in unforeseen ways.

These vulnerabilities can lead to serious consequences, such as the loss of funds or other assets, incorrect execution of contract terms, or even complete immobilization of a contract.

Reentrancy

This vulnerability occurs when attackers exploit the functionality of smart contracts and their interactions with external programs, enabling them to execute unauthorized code and reenter the program.

Reentrancy

An attacker can repeatedly call into a contract function and siphon away its Ether before the function has a chance to update its state.

A contract function calls an external function, which in turn calls back into the original function, thereby messing up its logic.

```java
public class BankAccount {
    private int balance;

    public BankAccount(int initialBalance) {
        this.balance = initialBalance;
    }

    public void withdraw(int amount) {
        if (balance >= amount) {
            notifyForWithdrawal(amount);  // pretend this method might call
            balance -= amount;
        }
    }

    public void notifyForWithdrawal(int amount) {
        // Some logic here that, let's say, under certain conditions ends u
        // Imagine this method is overridden in a subclass, or perhaps it's
        // acts upon the withdrawal.

        // For simplicity, let's just directly call `withdraw` again
        if (amount == 50) {
            withdraw(20);
        }
    }
}
```

```java
public class BankAccount {
    private int balance;

    public BankAccount(int initialBalance) {
        this.balance = initialBalance;
    }

    public void withdraw(int amount) {
        if (balance >= amount) {
            notifyForWithdrawal(amount);  // pretend this method might call
            balance -= amount;
        }
    }

    public void notifyForWithdrawal(int amount) {
        // Some logic here that, let's say, under certain conditions ends u
        // Imagine this method is overridden in a subclass, or perhaps it's
        // acts upon the withdrawal.

        // For simplicity, let's just directly call `withdraw` again
        if (amount == 50) {
            withdraw(20);
        }
    }
}
```

```java
public class Main {
    public static void main(String[] args) {
        BankAccount account = new BankAccount(100);

        account.withdraw(50);
        // What do you expect the balance to be?
    }
}
```

How can we fix it?

```
public class BankAccount {
    private int balance;
    private boolean isWithdrawing;

    public BankAccount(int initialBalance) {
        this.balance = initialBalance;
        this.isWithdrawing = false;
    }

    public synchronized void withdraw(int amount) {
        if (isWithdrawing) return;

        try {
            isWithdrawing = true;

            if (balance >= amount) {
                balance -= amount;
                notifyForWithdrawal(amount);
            }
        } finally {
            isWithdrawing = false;
        }
    }
```

Denial of Service (DoS)

DoS attacks render smart contracts ineffective either temporarily or permanently by employing various techniques, such as externally manipulated infinite loops or restricting specific operations to the contract’s owner.

```java
import java.util.concurrent.TimeUnit;

public class SimpleVotingSystem {
    private int candidateA = 0;
    private int candidateB = 0;

    public void voteForCandidateA() {
        validateVote();  // Simulate a delay for vote validation
        candidateA++;
    }

    public void voteForCandidateB() {
        validateVote();  // Simulate a delay for vote validation
        candidateB++;
    }

    private void validateVote() {
        try {
            TimeUnit.SECONDS.sleep(2);  // Simulate a time-consumin
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }

    public int getCandidateAVotes() {
        return candidateA;
    }

    public int getCandidateBVotes() {
        return candidateB;
    }
}
```

Imagine that someone wants to jam the voting system.

- They could rapidly fire **voteForCandidateA()** or **voteForCandidateB()** in a loop from multiple threads, making the system slow for everyone else.
- Since each vote takes a couple of seconds to validate, the system would be unable to process all incoming votes in a timely manner, effectively denying service to legitimate users.

DoS by Unbounded Operations: Some contracts have functions that loop through an array or perform operations that can grow in complexity depending on external factors. Attackers can exploit these functions by making them too costly in terms of gas to complete.

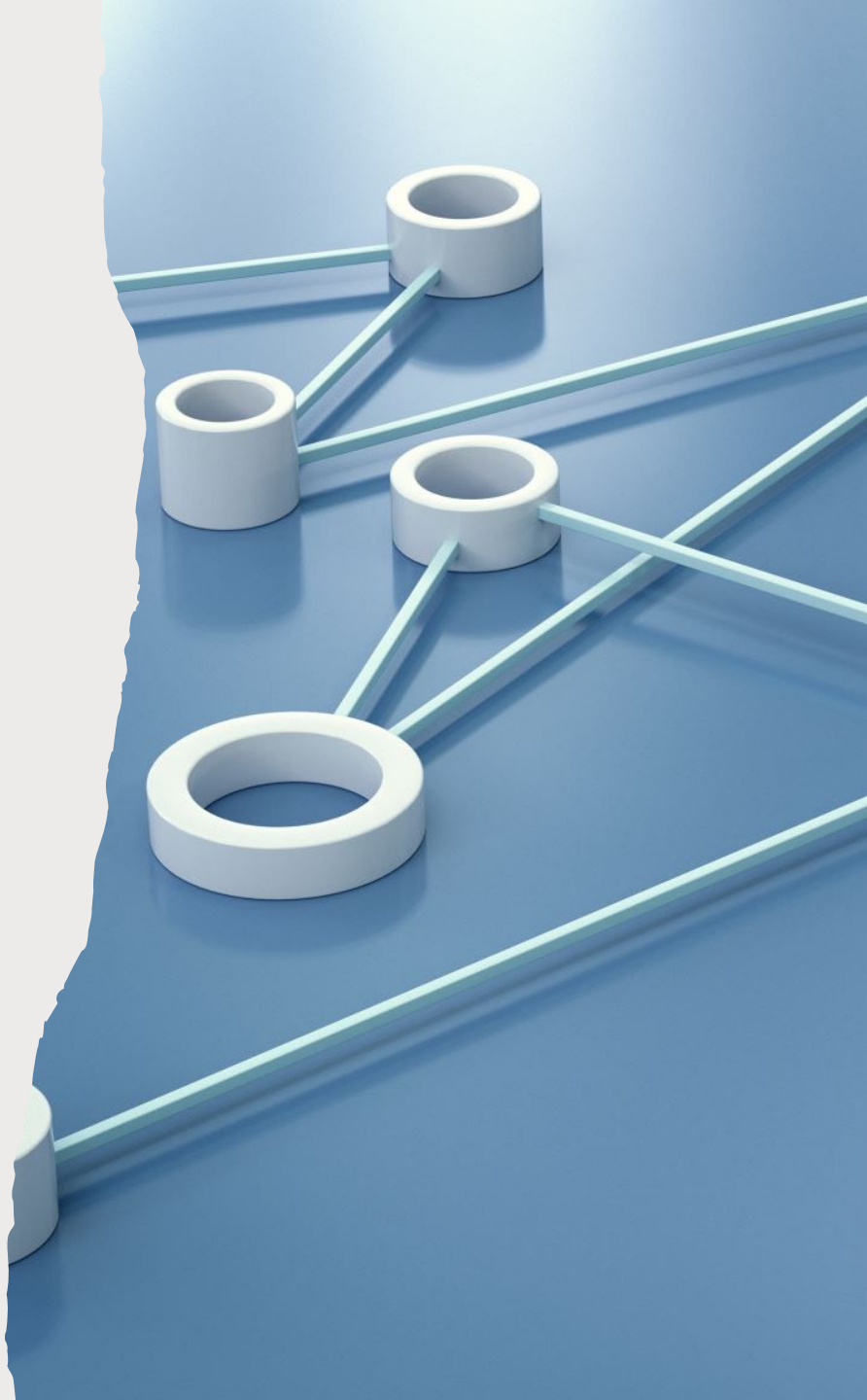
DoS by Gas Limit: Every transaction requires a certain amount of gas to execute. An attacker can create transactions that consume all the available gas, effectively blocking other transactions. Smart contracts that perform complex calculations or require a lot of storage are especially vulnerable to this kind of DoS attack.

DoS by Resource Manipulation: Another way to create a DoS condition is by using some dependency on external contracts or data to make a contract unusable. Consider a contract that has a dependency on another contract for some of its operations. If the external contract is compromised or becomes faulty, the dependent contract will also become unusable.

DoS by Stale Data: This is a sophisticated attack that exploits the delay between when a transaction is submitted and when it is added to the blockchain. If the contract's logic relies on some time-sensitive conditions, an attacker can manipulate these conditions in the time it takes for a transaction to be confirmed, effectively making the contract's operation faulty or leading it to an undefined state.

How to Defend Against DoS Attacks?

- **Time and Gas Awareness:** Always be aware of the computational complexity of your functions.
- **Rate Limiting:** Implement mechanisms to limit how frequently a function can be called.
- **Using Reputable External Contracts:** If your contract depends on external contracts, make sure they are reputable and secure.
- **Check-Effect-Interaction Pattern:** making sure you interact with external entities last can also help here.



Arithmetic Overflows and Underflows

Overflows and underflows can occur when fixed-size variables try to store numeric values or data that exceed their capacity.

```java
public class Main {
    public static void main(String[] args) {
        int maxInt = Integer.MAX_VALUE;  // 2^31 - 1
        int minInt = Integer.MIN_VALUE;  // -2^31

        System.out.println("Max integer value: " + maxInt);
        System.out.println("Max integer value + 1: " + (maxInt + 1));

        System.out.println("Min integer value: " + minInt);
        System.out.println("Min integer value - 1: " + (minInt - 1));
    }
}
```

When you run this code, you will see that adding 1 to **maxInt** will result in **minInt**, and subtracting 1 from **minInt** will give you **maxInt**.

The integer value wraps around because of the overflow and underflow, respectively.

```java
public class Main {
    public static void main(String[] args) {
        int maxInt = Integer.MAX_VALUE;
        int minInt = Integer.MIN_VALUE;

        if (maxInt == Integer.MAX_VALUE) {
            System.out.println("Overflow risk, can't add 1");
        } else {
            System.out.println("Safe to add 1: " + (maxInt + 1));
        }

        if (minInt == Integer.MIN_VALUE) {
            System.out.println("Underflow risk, can't subtract 1");
        } else {
            System.out.println("Safe to subtract 1: " + (minInt - 1));
        }
    }
}
```

How to Defend Against Overflows and Underflows

1

Bounds Checks: Always check boundaries before performing arithmetic operations.

2

Safe Math Libraries: Use libraries designed to handle these issues safely. For Solidity, the OpenZeppelin SafeMath library is a popular choice.

3

Up-to-date Language Versions: Use newer versions of programming languages that have built-in safeguards. For example, Solidity 0.8.x has built-in overflow and underflow checks.

Unchecked Low-Level Calls

This vulnerability is caused by incorrect use of the **call()** function, particularly when the validation of its return value is not adequately conducted.

```java
public class Calculator {
    public int divide(int a, int b) {
        return a / b;  // No checks for division by zero!
    }
}
```

```java
public class Main {
    public static void main(String[] args) {
        Calculator calculator = new Calculator();
        int result = calculator.divide(10, 0);  // This will throw an Arith
        System.out.println("Result: " + result);
    }
}
```

Fix (I)

```
public class Calculator {
    public int divide(int a, int b) throws ArithmeticException {
        if (b == 0) {
            throw new ArithmeticException("Division by zero");
        }
        return a / b;
    }
}
```

Fix (II)

```
public class Main {
    public static void main(String[] args) {
        Calculator calculator = new Calculator();
        try {
            int result = calculator.divide(10, 0);
            System.out.println("Result: " + result);
        } catch (ArithmeticException e) {
            System.out.println("Caught an exception: " + e.getMessage());
        }
    }
}
```

How to safeguard

Check the Return Value: Always check the return value when making low-level calls.

Use Safe Methods: Where possible, use higher-level abstractions that handle errors for you.

External Audits and Testing: Ensure that your contract undergoes thorough testing and possibly a security audit.

Time Dependency

Contracts often use timestamps within critical functions related to the transfer of funds.

However, the **’block.timestamp’** variable can be manipulated by a miner, indicating the need for careful usage.

```java
public class Auction {
    private int highestBid;
    private long auctionEndTime;

    public Auction() {
        this.highestBid = 0;
        this.auctionEndTime = System.currentTimeMillis() + 60000; // 60 se
    }

    public void bid(int amount) {
        if (System.currentTimeMillis() > auctionEndTime) {
            System.out.println("Auction is already over.");
            return;
        }

        if (amount > highestBid) {
            highestBid = amount;
            System.out.println("New highest bid: " + highestBid);
        }
    }
public int finalizeAuction() {
    if (System.currentTimeMillis() <= auctionEndTime) {
        System.out.println("Auction is not yet over.");
        return -1;
    }

    System.out.println("Auction is over. Final highest bid: " + highestBid);
    return highestBid;
}
```

- In this simple example, the bid is only accepted if the current system time is before the auction's end time (**auctionEndTime**).
- The auction's result is finalized only if the current system time is after **auctionEndTime**.

This naive time check can be exploited in various ways:

- If the system clock is adjusted backward, bidders could potentially make bids even after the auction was supposed to have ended.
- If the system clock is adjusted forward, the auction could end prematurely.

How to address this in real systems?

- In a real system, you'd want to have more robust time checks. For example, you might rely on multiple, trusted time sources rather than just the system clock.

- In Ethereum smart contracts, the issue is more complex due to the decentralized nature of the blockchain. You can't control or fully trust the system time. That's why relying on block numbers instead of timestamps, or using external time-oracles might be safer alternatives.

Bad Randomness

- Randomness is a crucial factor in many systems, especially in secure transactions or games.
- The issue is that getting truly random numbers in a deterministic system like a blockchain—or even in most programming languages—is pretty challenging.
- Developers often resort to pseudo-random generation as a workaround.

```java
public class Lottery {
    public static void main(String[] args) {
        // Assume we have 10 players with IDs 1 to 10.
        int winner = (int) (Math.random() * 10) + 1;
        System.out.println("The winner is: " + winner);
    }
}
```

In this example,

Math.random() generates a pseudo-random number. However, the algorithm used by Math.random() is not suitable for cryptographic or high-security needs because:

- The random number generator's initial state could potentially be guessed or known.
- The random number generation process might be observable, allowing someone to predict future outputs.
- These issues could be exploited in a way that allows an attacker to have a higher chance of winning the lottery than what would be expected in a fair system.

```java
import java.security.SecureRandom;

public class SecureLottery {
    public static void main(String[] args) {
        SecureRandom secureRandom = new SecureRandom();

        // Assume we have 10 players with IDs 1 to 10.
        int winner = secureRandom.nextInt(10) + 1;

        System.out.println("The secure winner is: " + winner);
    }
}
```

The Right Tools for the Job

- in a smart contract, you might opt for a dedicated, external source of randomness like Chainlink VRF (Verifiable Random Function), rather than relying on something like block hashes or timestamps.

- "bad randomness" vulnerabilities happen when the source of randomness used isn't random enough for the application's needs, allowing for potential manipulation or prediction.

Safeguarding Against Bad Randomness

- **Use a Trusted Oracle:** For critical functions requiring randomness, a trusted off-chain source may be necessary.
- **Commit-Reveal Schemes:** In this approach, users commit to a secret number, then reveal it later. Combining all revealed numbers gives a more random value.
- **Use Multiple Sources of Data:** Sometimes, multiple less-reliable sources can together provide a more reliable random number.

Access control

This vulnerability pertains to the unauthorized access to structs, variables, and functions that should ideally be accessible only to specific users.

```java
public class SecureVault {
    private int secretCode = 1234;

    public void openVault(int code) {
        if (code == secretCode) {
            System.out.println("Vault opened!");
        } else {
            System.out.println("Wrong code! Cannot open vault.");
        }
    }
}
```

```java
public class Main {
    public static void main(String[] args) {
        SecureVault vault = new SecureVault();

        // Assume user input is collected and stored in this variable
        int userInput = 1234;

        vault.openVault(userInput);
    }
}
```

```java
public class SecureVault {
    private int secretCode = 1234;
    private int attempts = 0;

    public void openVault(int code) {
        if (attempts >= 3) {
            System.out.println("Too many failed attempts. Access locked.");
            return;
        }

        if (code == secretCode) {
            System.out.println("Vault opened!");
            attempts = 0;
        } else {
            attempts++;
            System.out.println("Wrong code! Cannot open vault.");
        }
    }
}
```

- In Ethereum smart contracts, access control is often managed through modifiers that restrict which addresses can call certain functions.
- A lack of or improper use of such modifiers can result in similar vulnerabilities where unauthorized addresses can execute functions they shouldn't be able to.
- The key to mitigating access control vulnerabilities is careful design, often involving multiple layers of checks and authentication.

Front Running

This issue arises from the mining process. The time taken to mine transactions allows an attacker to send a transaction and include it in a block before the original transaction.

Front-running in the context of blockchain and smart contracts typically occurs because someone sees a pending transaction and then pays a higher gas fee to have their own transaction processed first.

How to Mitigate:

- **Commit-Reveal:** Users first commit to making a trade without revealing details and then reveal the trade information later for execution.
- **Timelocks:** Enforce a waiting period for important actions to take place, giving everyone a fair chance to act.
- **Randomization:** Some DeFi protocols use on-chain or off-chain mechanisms to add a degree of randomization to the order in which transactions are processed.

Short Address

A Short Address Attack occurs when an attacker sends a transaction with fewer bytes than expected for an Ethereum address.

The smart contract, following the behavior of the Ethereum Virtual Machine (EVM), automatically pads the missing bytes with zeros, potentially redirecting the transaction to an unintended address.

Preventing Short Address Attacks involves careful coding on the part of the smart contract developers as well as diligent input validation by user interfaces and applications that interact with the contract.

On the SC side

- **Input Validation:** Explicitly check that the length of the address is as expected. However, keep in mind that Solidity naturally pads shorter addresses with zeros, so this might be hard to enforce in the contract itself.
- **Safe Libraries:** Use well-tested libraries designed to handle token transfers and other operations securely.
- **Revert on Error:** Make sure to use require or revert to stop the transaction if something is wrong.

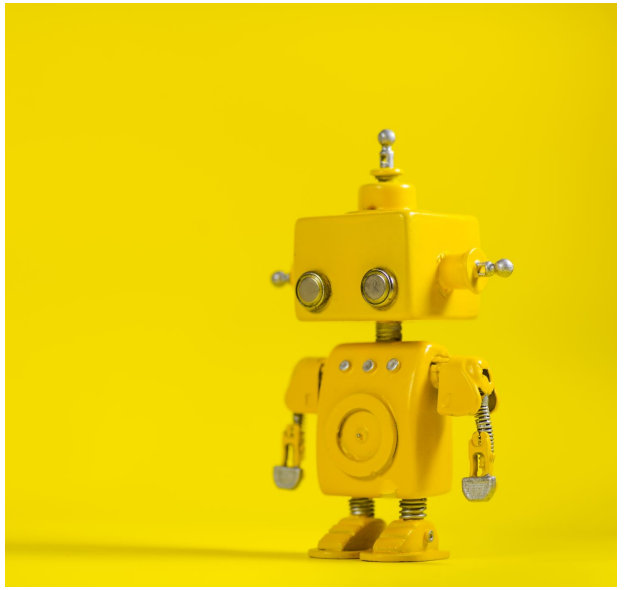
On the User Interface and Off-Chain Side

- **Client-Side Validation:** The web interface or application should validate that addresses are of the correct length and structure before even sending the transaction.
- **User Confirmation:** Always allow the user to review and confirm the details of a transaction before it is sent, including the full address to which funds will be transferred.
- **Secure Communication:** Use secure and verified libraries for crafting and sending raw transactions to ensure that the data is correctly formatted.
- **Warnings:** Implement warning messages that can alert users if they are about to interact with a contract that does not implement standard security checks.
- **Third-Party Audits:** Getting your code audited by a reputable security firm can also uncover potential vulnerabilities like these.

Automated Program Repair

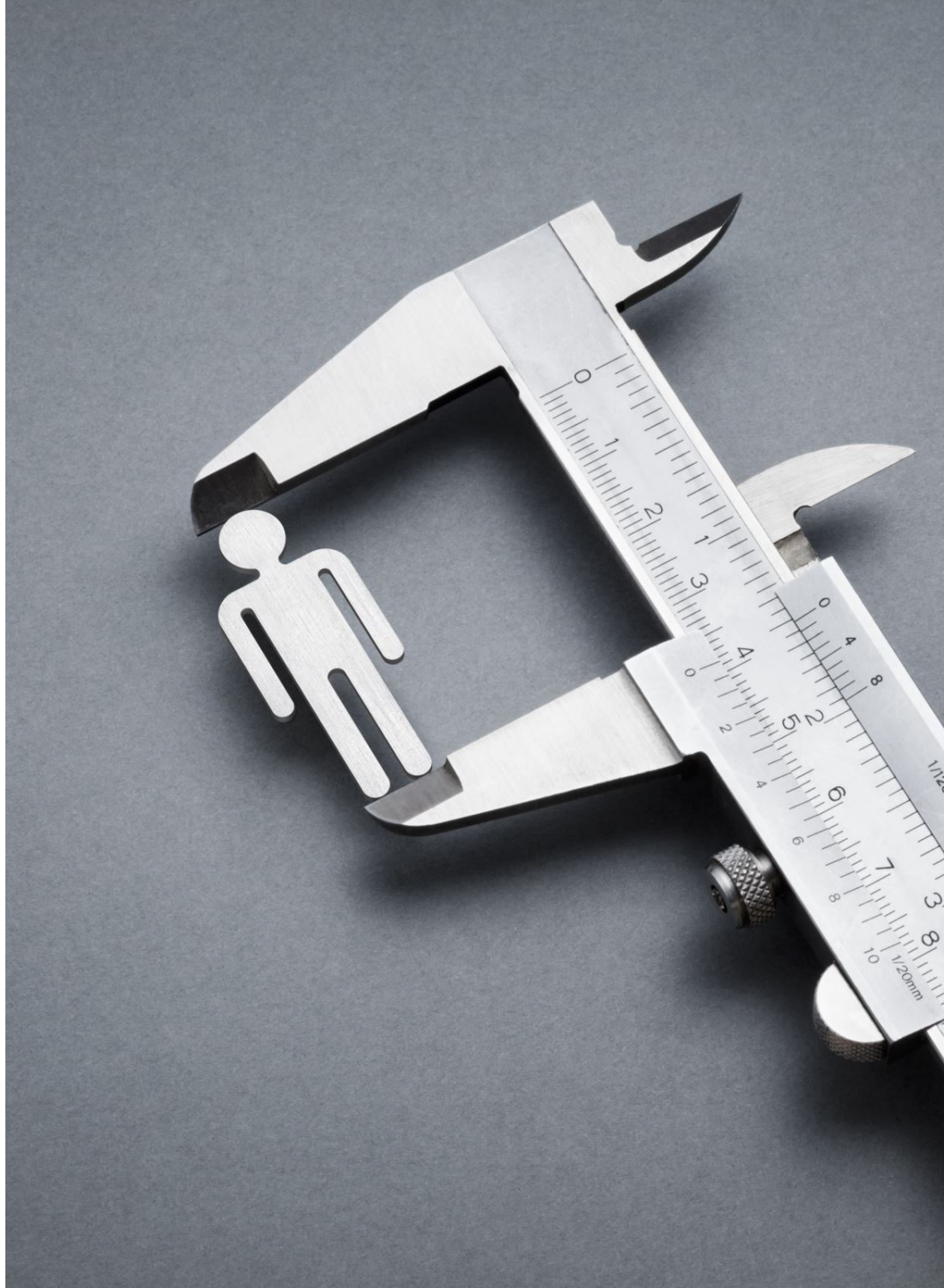
We explored new methods for Automated Program Repair (APR) of smart contracts using Natural Language Processing (NLP) techniques.

We investigated the potential use of OpenAI's models, such as ChatGPT, to automatically repair Solidity programs and compares this to Google's Bard.

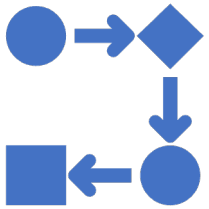


We used three methods:

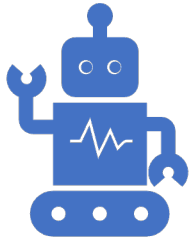
- The direct input of SCs, with the models expected to identify and rectify potential vulnerabilities. Here, ChatGPT and Bard serve as vulnerability detection and APR tools.
- The input of a test subset of SCs, including known vulnerabilities and their fixes. This approach involves training ChatGPT and Bard using supervised learning and then testing them with another subset of SCs.
- The input of SCs that include known vulnerabilities and the corresponding affected code lines. In this case, ChatGPT and Bard serve only as APR tools, with vulnerabilities identified by dedicated tools.



ways to automatically fix these issues using AI language models



Giving the contract to the AI and letting it find and fix issues.

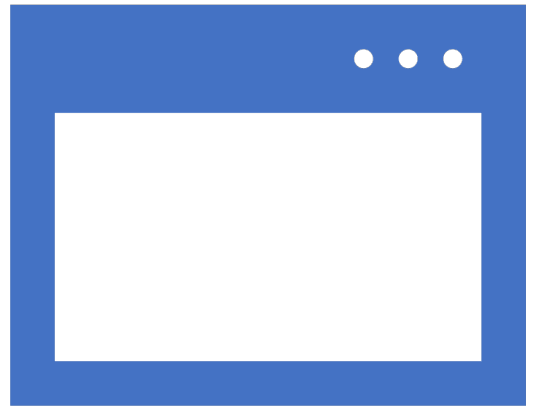


Training the AI on a subset of contracts that have known issues and fixes.

Using the AI only to fix issues that another tool has already found.

Sample

- Short Address: One instance is susceptible to the short address attack.
- Arithmetic overflows and underflows: We have 105 LOC exposed to arithmetic overflows and underflows.
- Access Control: 18 LOC are vulnerable to access control.
- Bad Randomness: Bad randomness affects 8 LOC.
- Denial Of Services: 7 LOC could lead to denial of service.
- Front Running: 4 instances are exposed to front-running attacks.
- Reentrancy: In our dataset, 90 possible reentrant patterns are included.
- Time Manipulation: 5 examples that could lead to attacks due to time manipulation.
- Unchecked Low Level Calls: 56 lines of code could lead to attacks due to ULLC.



Patching process

- We used a supervised learning strategy to train the model.
- This process involved supplying the model with examples of vulnerable code snippets alongside their corresponding fixes for a selection of each vulnerability type.
- To ensure the effectiveness of the provided fixes, we conducted re-analysis of the smart contracts using three diagnostic tools.
- If these tools did not identify any vulnerabilities, we considered the fix to be correct.

Prompt Engineering Process

- The first method involves providing ChatGPT and Bard with both the buggy contract and the corresponding patched version.
- After supplying the model with a substantial sample of examples covering various vulnerabilities, we can evaluate its performance by asking it to fix a vulnerable contract without providing the patch.
- The second method utilizes ChatGPT and Bard both as a diagnostic tool to identify vulnerabilities and as an Automatic Program Repair tool.

Prompt Engineering Process

- The third method combines ChatGPT and Bard with ad-hoc diagnostic tools specifically de- signed for smart contract vulnerability detection.
- The first phase involves running tools to scan a substantial sample, annotating the vulnerable lines with the corresponding vulnerabilities.
- The second phase provides ChatGPT and Bard with the vulnerable contract and the annotations derived from the diagnostic process. We indicated the vulnerable code line and the associated vulnerability, explicitly asking the models to fix the contract while maintaining the given pragma version, except in cases where version updates are necessary.

Criteria for Correct Output

A contract is correct if:

- All its security issues are fixed
- It can still be compiled (turned into a program that can run)

A contract is incorrect if any of the following happen:

- It still has security flaws
- Only some flaws are fixed, but not all
- The program doesn't compile anymore
- The version info of the contract is changed

How Success is Measured

Smart contracts are sorted into three types based on the fixes:

- Non-Compilable (NC): The contract still has errors and can't be compiled.
- Non-Analyzable (NA): The tool didn't even look at the contract. So, the security issues remain.
- Successful (S): The contract is fine. All issues fixed, and it can be compiled.

ChatGPT as an APR Tool with Training Samples

Category	Correct ChatGPT	Incorrect ChatGPT
Short Address	*	*
Overflow and Underflow	41	24
Front Running	1	0
Reentrancy	35	12
Denial of Service	3	0
Unchecked Low-Level Calls	18	1
Bad Randomness	3	1
Time Manipulation	2	0
Access Control	5	1

Table 12: Number of fixed exposures in the testing session by ChatGPT

Category	Correct Bard	Incorrect Bard
Short Address	*	*
Overflow and Underflow	13	52
Front Running	1	0
Reentrancy	26	21
Denial of Service	3	0
Unchecked Low-Level Calls	7	12
Bad Randomness	4	0
Time Manipulation	2	0
Access Control	3	3

Table 14: Number of fixed exposures in the testing session by Bard

ChatGPT as a Vulnerability Detection and APR Tool

Category	NC	NA	S
Short Address	0	0	1
Overflow and Underflow	3	0	13
Front Running	0	0	4
Reentrancy	2	0	29
Denial of Service	1	0	5
Unchecked Low-Level Calls	0	2	49
Bad Randomness	0	1	7
Time Manipulation	0	0	5
Access Control	0	0	18
Multi-Vulnerable	7	4	49

Table 6: Number of Non-Compilable, Non-Analyzable and Successful Smart Contracts Samples in the Chat-GPT Vulnerabilities Detection and APR Process

Category	NC	NA	S
Short Address	0	1	0
Overflow and Underflow	6	2	7
Front Running	2	0	2
Reentrancy	0	5	26
Denial of Service	0	2	4
Unchecked Low-Level Calls	29	13	9
Bad Randomness	0	4	4
Time Manipulation	0	3	2
Access Control	2	4	12
Multi-Vulnerable	6	26	28

Table 8: Number of Non-Compilable, Non-Analyzable and Successful Smart Contracts Samples in the Bard Vulnerabilities Detection and APR Process

ChatGPT and Bard as an APR Tool with Exposed Lines and Associated Vulnerabilities

Category	NC	NA	S
Short Address	0	0	1
Overflow and Underflow	0	0	16
Front Running	0	0	4
Reentrancy	1	0	30
Denial of Service	0	0	6
Unchecked Low-Level Calls	1	3	47
Bad Randomness	0	0	8
Time Manipulation	0	0	5
Access Control	0	0	18
Multi-Vulnerable	3	2	55

Table 1: Number of Non-Compilable, Non-Analyzable and Successful Smart Contracts Samples in the Chat-GPT APR Process

Category	NC	NA	S
Short Address	1	0	0
Overflow and Underflow	10	1	5
Front Running	0	0	4
Reentrancy	26	5	0
Denial of Service	1	2	3
Unchecked Low-Level Calls	13	13	25
Bad Randomness	3	4	1
Time Manipulation	1	2	2
Access Control	3	5	10
Multi-Vulnerable	14	4	42

Table 3: Number of Non-Compilable, Non-Analyzable and Successful Smart Contracts Samples in the Bard APR Process

Conclusion

- Our analysis found that Chat- GPT consistently performed better than Google's Bard across all setups. While Bard successfully addressed some vulnerabilities, it exhibited certain limitations in its capacity for automated smart contract repair.
- Our results indicate that the most effective methodology involves providing the models with the vulnerable code snippet, listing all the exposed lines of code, and specifying the associated vulnerabilities. This approach achieved an accuracy rate of 89%, successfully repairing the majority of smart contracts.
- It appears that Chat- GPT and Bard can be effectively utilized as APR tools for smart contracts when used in tandem with specific vulnerability detection applications.
- Our study also identified limitations related to the tools' ability to analyze large contracts. This issue could be partially mitigated by segmenting larger contracts into smaller parts for analysis, but ideally, the tools should be able to scan entire contracts in one operation.