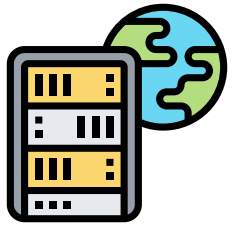
Front-running, Imitation, Decompilation

Dark Forest Survival Skills

Kaihua Qin
University of Warwick

Recap

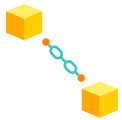


World Computer

Smart Contract
(Program running on blockchains)

- Non-custodial
- Transparent
- Censorship Resistant
- . . .

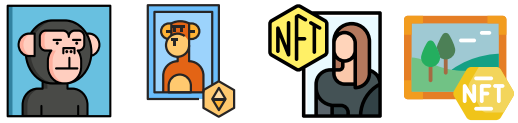
Decentralized Finance
(DeFi)



Public Ledger

Distributed Consensus

Peer-to-peer Network



Non-Fungible Tokens (NFTs)

Exchange

Lending

Governance

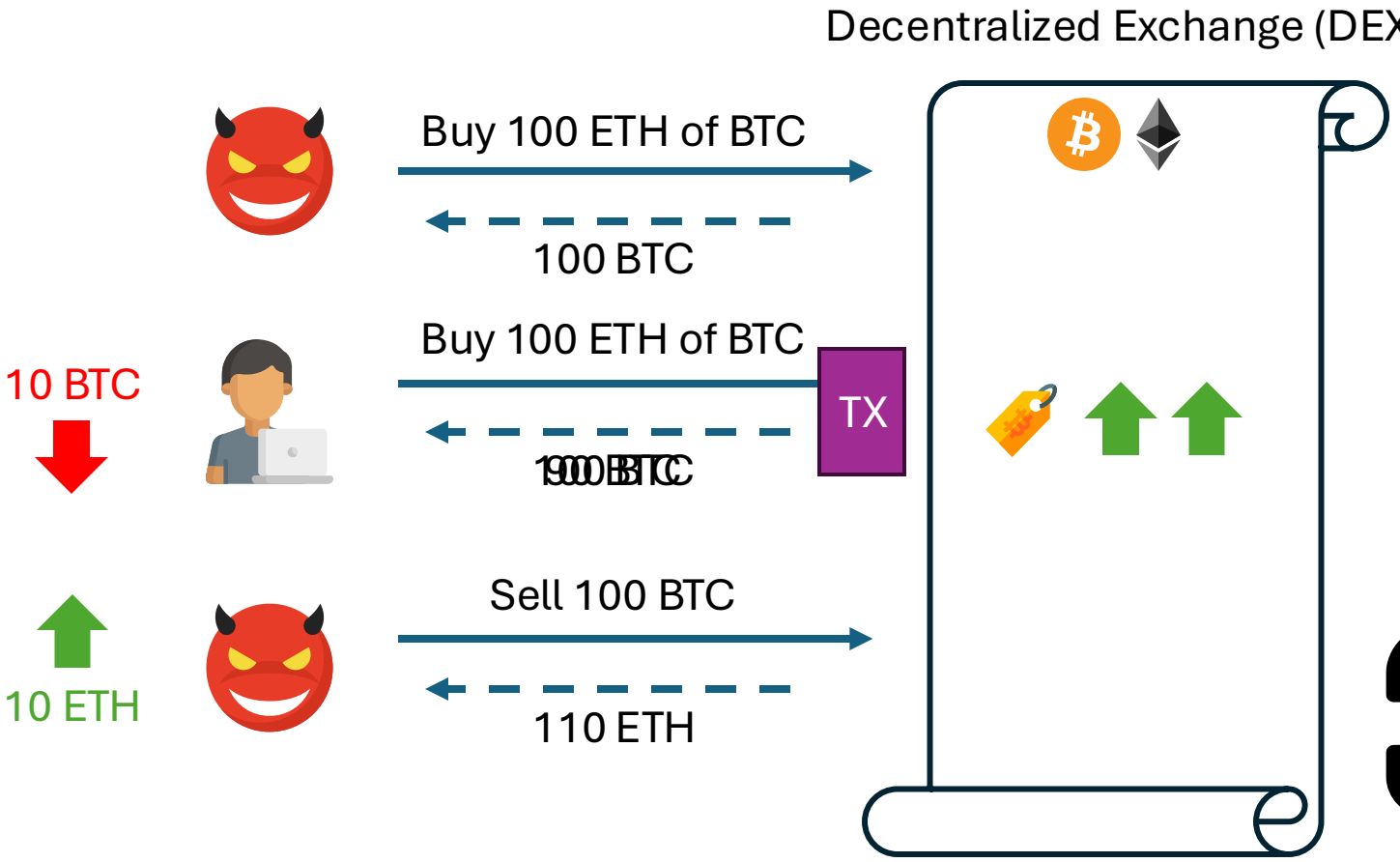
Identity

Stablecoin

• • •

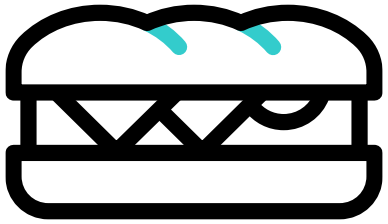

Ethereum is
THE DARK
FOREST

Sandwich Attack



1. Transactions are publicly visible

2. Transaction ordering is centralized
   - Builders – monopoly
   - Bribery

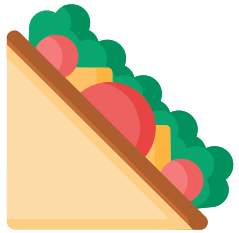


Front-running
Victim
Back-running

Quantifying Miner/Maximal Extractable Value (MEV)

Arbitrage

Liquidation



Sandwich

- **Dataset: the public Ethereum blockchain**
  - 11,289 unique addresses
  - 49,691 cryptocurrencies
  - 60,830 on-chain markets
- **Over 32 months (December 2018 – August 2021)**
  - About 540M USD in profit
  - Highest instance: 616.6x the Ethereum block reward

Qin K, Zhou L, Gervais A. Quantifying blockchain extractable value: How dark is the forest?. In 2022 IEEE Symposium on Security and Privacy (SP) 2022 May 22 (pp. 198-214). IEEE.

EigenPhi

Ethereum

Search by Address/Txn Hash/Block Number/Symbol

Reports Docs

Tools

Market Overview

24H 7D 30D

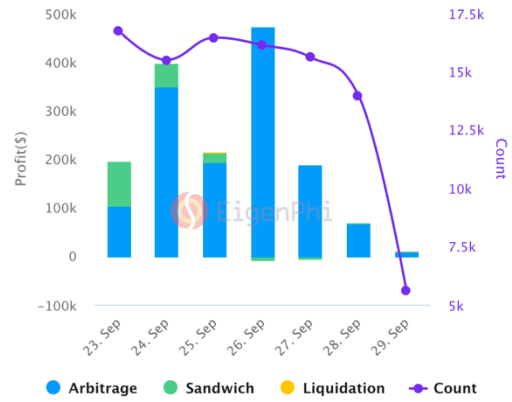
Performance of MEV Types

By Profit By Volume

Arbitrage(7D)
$1.78M

Sandwich(7D)
$146.06k

Liquidation(7D)
$11.31k



21 minutes ago

MEV Live-Stream

More

		Contract	Profit	Cost
#23467837 secs	Arbitrage	0x814...b5128	$0.31	$1.26
#23467836 secs	Arbitrage	0x22A...75622	$0.21	$0.87
#23467836 secs	Arbitrage	0x814...b5128	$0.50	$2.00
#23467836 secs	Arbitrage	0x814...b5128	$0.13	$2.76
#23467835 secs	Arbitrage	0x657...65B33	$0.05	$0.47
#23467835 secs	Arbitrage	0x073...25479	$0.28	$1.15

MEV Contract Profit Leaderboard

ALL Types Arbitrage Sandwich Liquidation

0xbd9...41415
Arbitrage
Profit: $565,725.39
Cost: $43.65

0x65c...a194B
Arbitrage
Profit: $512,367.49
Cost: $36.79

0xF3D...0F0F9
Arbitrage
Profit: $162,394.31
Cost: $1,712.90

0x285...9a59B
Arbitrage
Profit: $116,101.37
Cost: $22,023.61

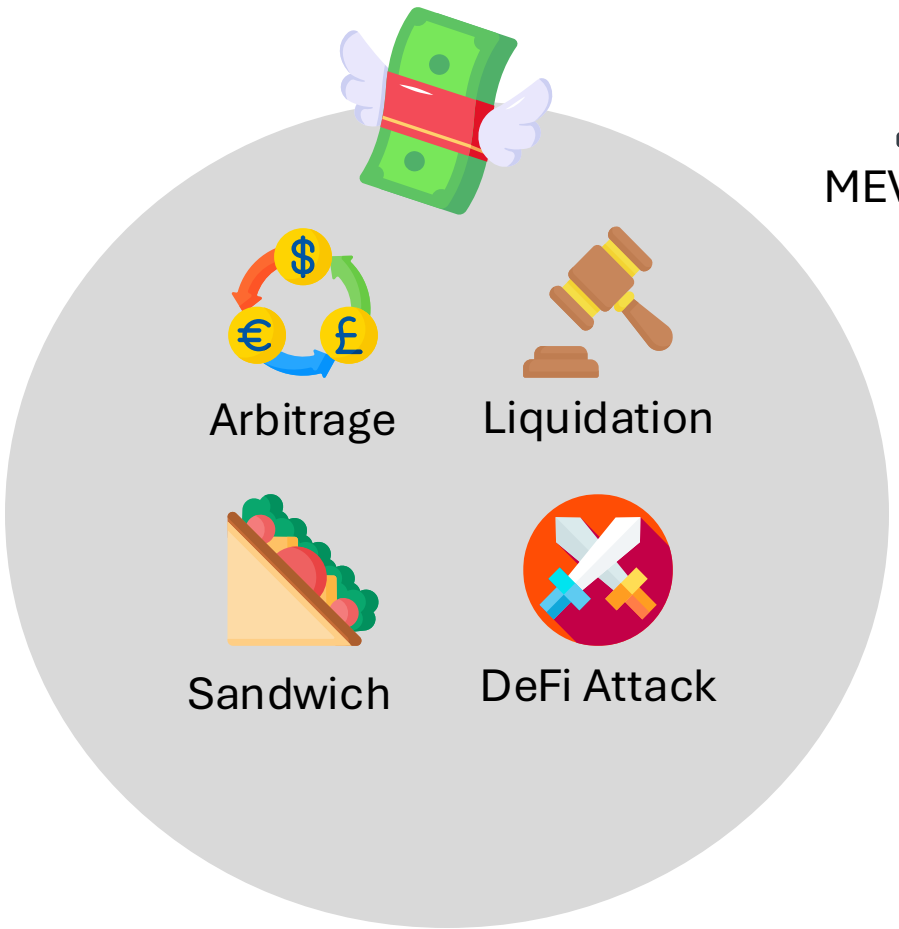
0x1f2...Df387

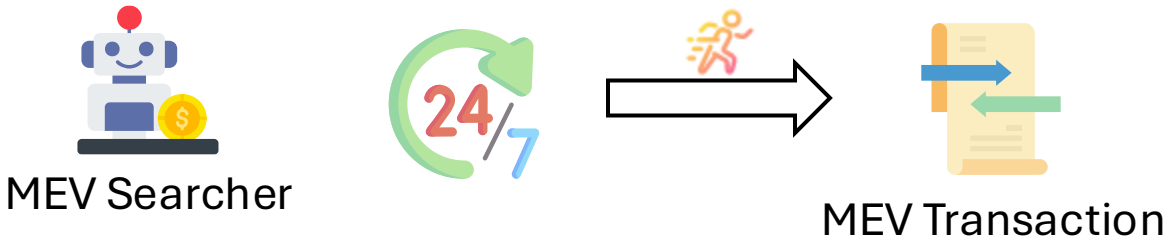
0x000...16B40
Arbitrage Sandwich

0xFF5...1f78f
Arbitrage

0x7a2...2488D
UNISWAP
Sandwich

MEV Extraction

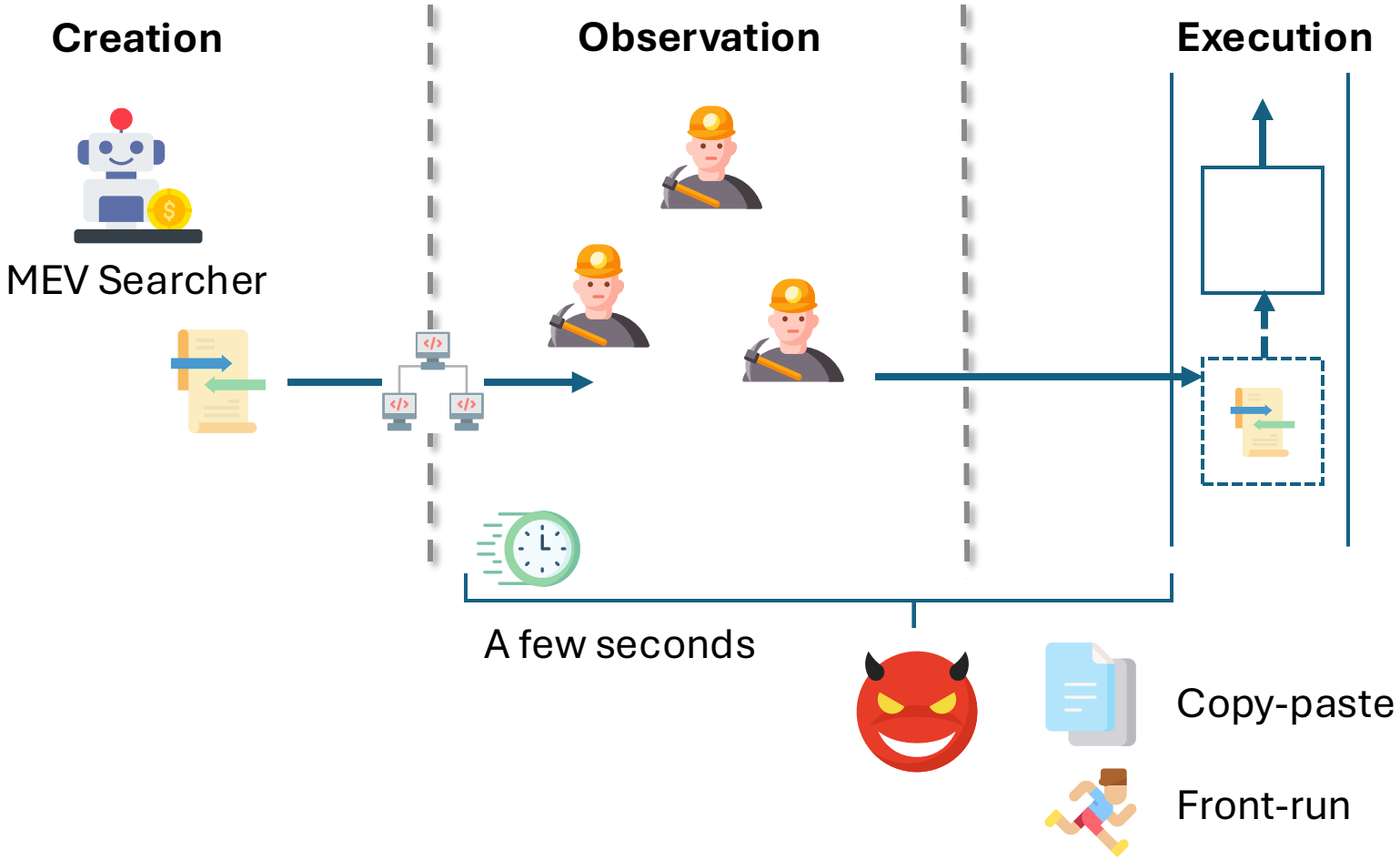





```
contract MEV {
  function arb(uint x, uint y) public {
    swapETHtoBTC(x);
    swapBTCtoETH(y);
    msg.sender.transfer(profit);
  }
}
```

The Blockchain Imitation Game

Imitation

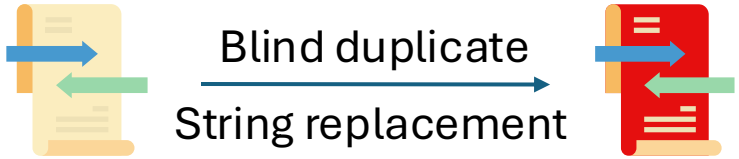

Creation
Observation
Execution
MEV Searcher
A few seconds
Copy-paste
Front-run

Naive Imitation

Potential Transaction

Step 1

Imitation Transaction




```
1 contract MEV {
2   function arb(uint x, uint y) public {
++    require(msg.sender==0x12..);
3     swapETHtoBTC(x);
4     swapBTCtoETH(y);
5     msg.sender.transfer(profit);
6   }
7 }
```

Step 2
Execute and validate locally

Step 3
Front-run if profitable

Simple but effective

Ethereum

December 2018 – August 2021

35M USD

Easy to prevent

Generalized Imitation

The threat of imitation remains

MEV Contract

```
1  contract MEV {
2    function arb(uint x, uint y) public {
3
4      require(msg.sender==0x12..);
5
6      swapETHtoBTC(x);
7      swapBTCtoETH(y);
8      msg.sender.transfer(profit);
9    }
10 }
```

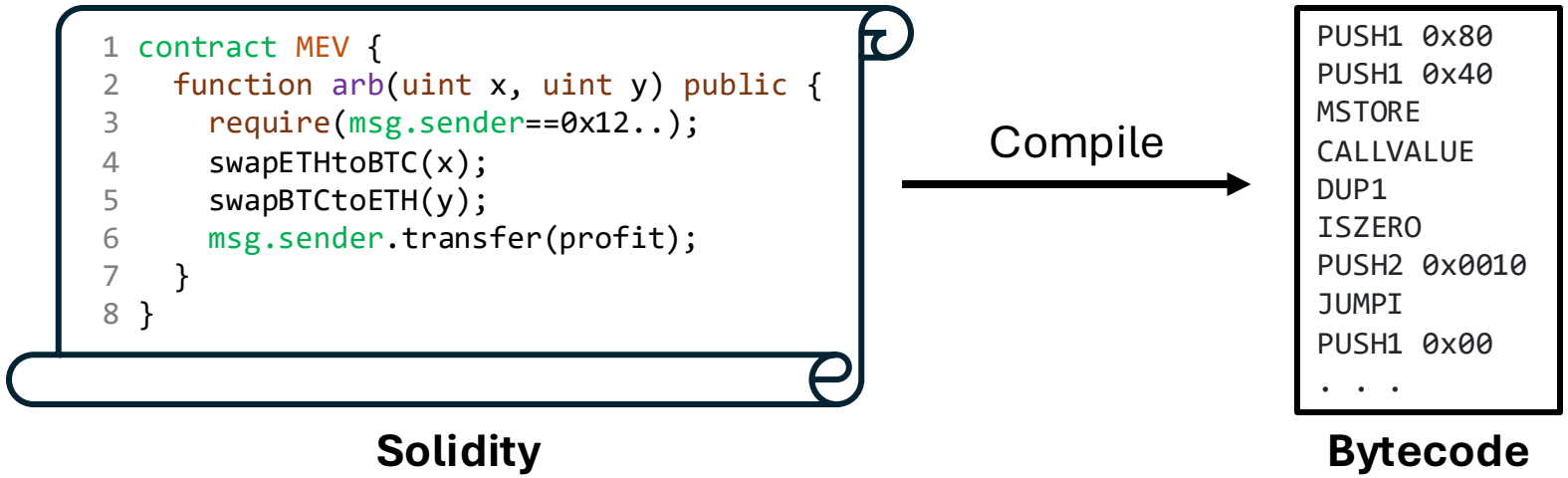
Synthesize

Imitation Contract

```
1  contract MEV {
2    function arb(uint x, uint y) public {
3
4      require(msg.sender==0x12..);
5
6      swapETHtoBTC(x);
7      swapBTCtoETH(y);
8      msg.sender.transfer(profit);
9    }
10 }
```

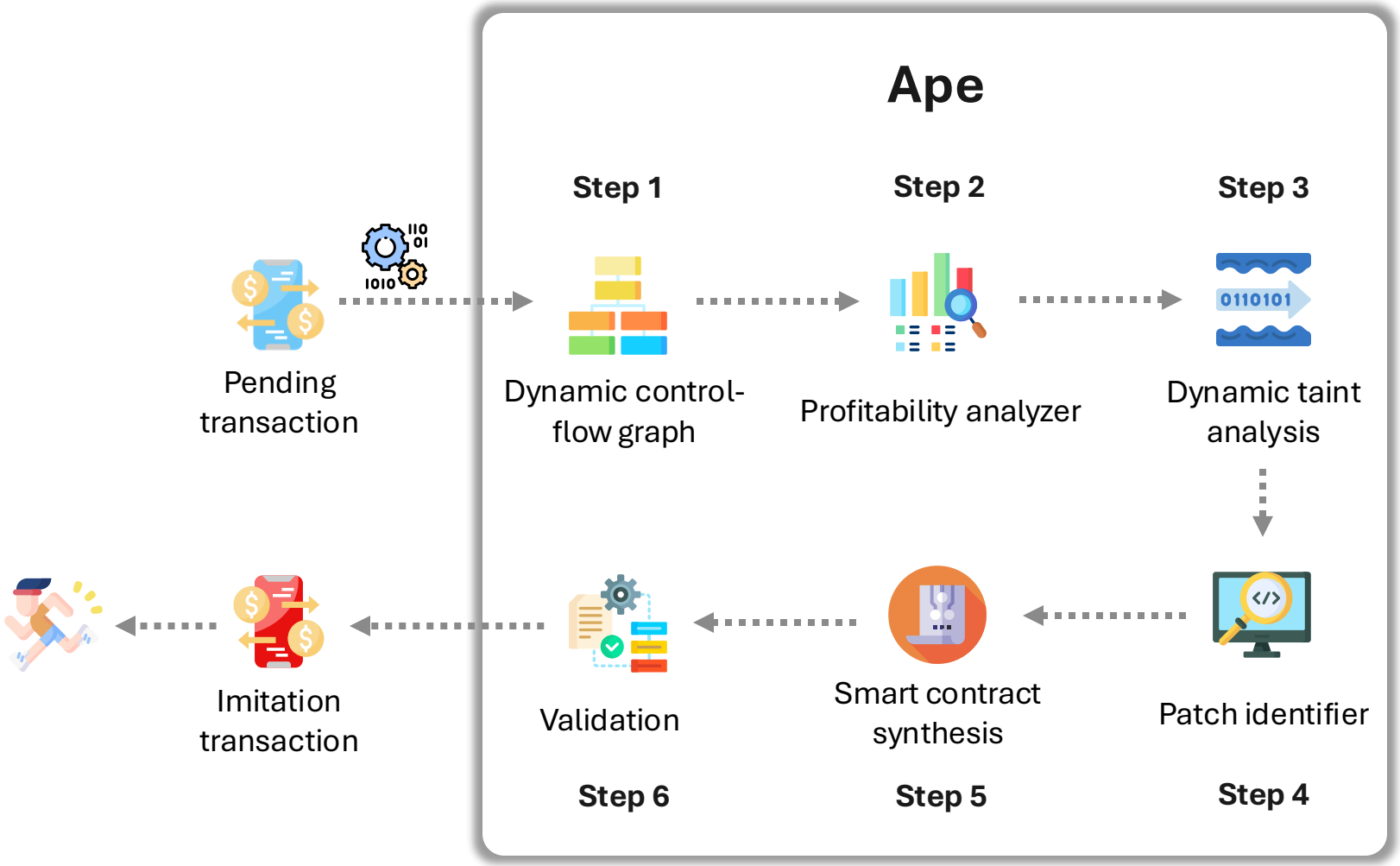
Qin K, Chaliasos S, Zhou L, Livshits B, Song D, Gervais A. The blockchain imitation game. (2023). In 32nd USENIX Security Symposium (USENIX Security 23) (pp. 3961-3978).

Technical Challenges



- Only low-level bytecode is available
- High-level application-layer semantics are missing
- Short time window (0.75 – 12 seconds)

Ape Overview



Dynamic control-flow graph

- Graphical representation of transaction and smart contract execution

Dynamic taint analysis

- Identify imitation protection w/o high-level semantics
- Virtual machine instrumentation ensuring O(1) complexity

Smart contract synthesis

- Automated

Force to jump

```
...
+ SWAP1
+ POP
+ JUMP
- JUMPI
...
```

Force not to jump

```
...
+ POP
+ POP
- JUMPI
...
```

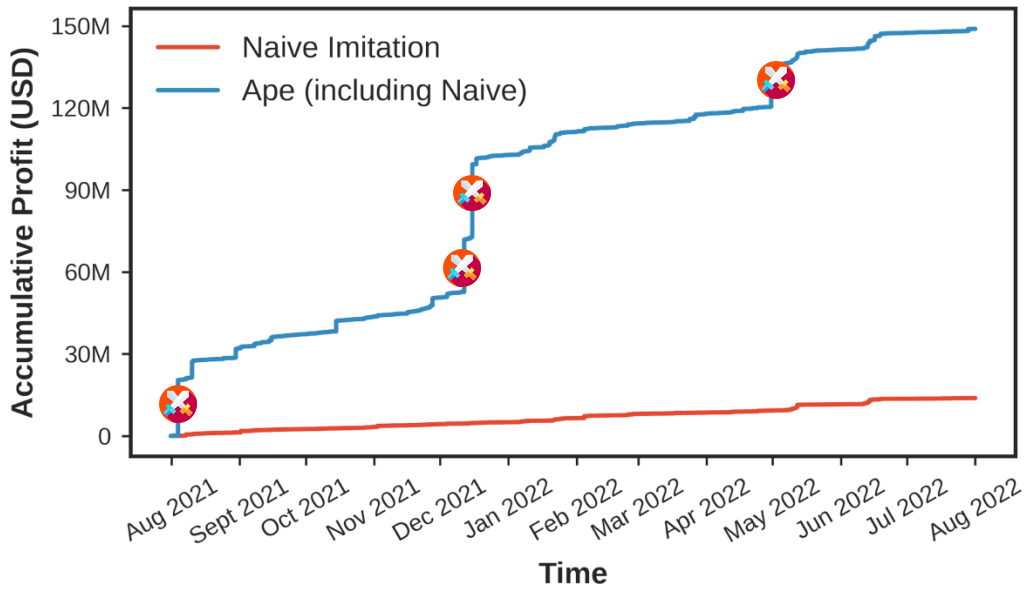
Ape Evaluation

August 1, 2021 – July 31, 2022 (1 year)

0.07 second

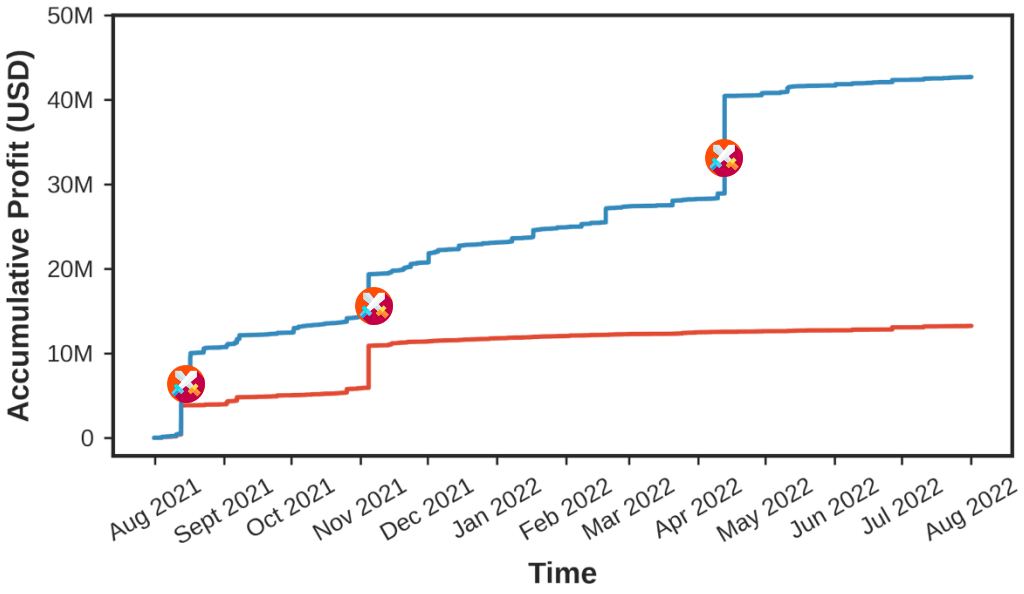
Ethereum

148.96M USD

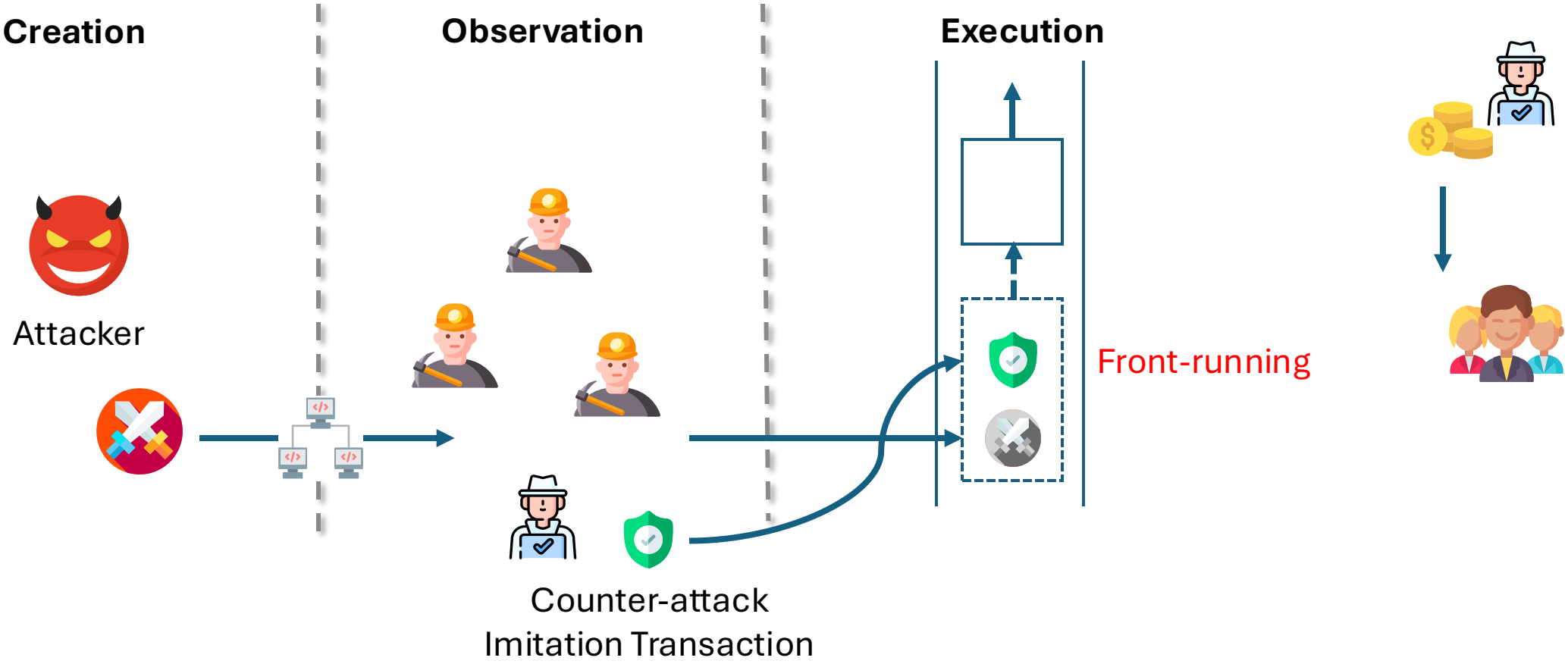


BSC

42.70M USD



Imitation as Whitehat



Imitation as Whitehat

Preventable DeFi Attacks (August 1, 2021 – July 31, 2022)

Ethereum

29 73.74M USD

Protocol	Loss (USD)	Date
Popsicle Finance	20.25M	Aug-03-2021
Saddle Finance	9.71M	Apr-30-2022
Indexed Finance	3.58M	Oct-14-2021
…	…	…

BSC

40 22.39M USD

Protocol	Loss (USD)	Date
Elephant Money	11.52M	Apr-12-2022
XSURGE	5.17M	Aug-16-2021
CollectCoin	1.06M	Dec-01-2021
…	…	…

The Blockchain Imitation Game

Kaihua Qin
Imperial College London, RDI

Stefanos Chaliasos
Imperial College London

Liyi Zhou
Imperial College London, RDI

Benjamin Livshits
Imperial College London

Dawn Song
UC Berkeley, RDI

Arthur Gervais
University College London, RDI

Abstract

The use of blockchains for automated and adversarial trading has become commonplace. However, due to the transparent nature of blockchains, an adversary is able to observe any pending, not-yet-mined transactions, along with their execution logic. This transparency further enables a new type of adversary, which copies and front-runs profitable pending transactions in real-time, yielding significant financial gains.

Shedding light on such "copy-paste" malpractice, this paper introduces the Blockchain Imitation Game and proposes a generalized imitation attack methodology called APE. Leveraging dynamic program analysis techniques, APE supports the automatic synthesis of adversarial smart contracts. Over a timeframe of one year (1st of August, 2021 to 31st of July, 2022), APE could have yielded 148.96M USD in profit on Ethereum, and 42.70M USD on BNB Smart Chain (BSC).

Not only as a malicious attack, we further show the potential of transaction and contract imitation as a defensive strategy. Within one year, we find that APE could have successfully imitated 13 and 22 known Decentralized Finance (DeFi) attacks on Ethereum and BSC, respectively. Our findings suggest that blockchain validators can imitate attacks in real-time to prevent intrusions in DeFi.

1 Introduction

Decentralized Finance (DeFi), built upon blockchains, has grown to a multi-billion USD industry. However, blockchain peer-to-peer (P2P) networks have been described as dark

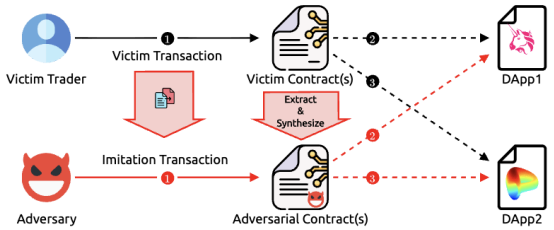


Figure 1: High-level APE attack mechanism, a generalized, automated imitation method synthesizing adversarial contracts without prior knowledge about the victim's transaction and contract(s). APE appropriates any resulting revenue.

front-run a pending profitable transaction without understanding its logic. We term such a strategy as an *imitation attack*. A naive string-replace imitation method [46] was shown to yield thousands of USD per month on past blockchain states. The practitioners' community swiftly came up with defenses to counter such a naive imitation method. At the time of writing, traders often deploy personalized and closed-source smart contracts, making exploitation harder. The known naive imitation algorithm no longer applies, because these contracts are typically protected through, for example, authentications.

However, the possibility of a generalized imitation attack that can invalidate existing protection mechanisms has not yet been explored. The goal of this work is to investigate, design, implement, and evaluate a generalized imitation method.

Looking into an MEV Contract

Bytecode

```
0x60406080815260048036101561004e575b50361561001c57600080fd5b513381
523460208201527f88a5966d370b9919b20f3e2c13ff65706f196a4e32cc2c12bf
57088f8852587490604090a1005b6000803560e01c91826301e3366714610b4057
82631878068414610ae757826323a69e751461010257826329 4b038d14610a9857
82632c8958f6146101025782633b534c6614610884578263 3ccfd60b1461083257
82633e88c8ab1461076b5782637c453caa1461073e578263923b8a2a1461070c57
8263a1dab4eb14610102578263d3e1c284146106bb578263fa09e6301461010757
50508063fa461e33146101025763fa483e721461010257386100105656b610ca656
5b83346103e357602080600319360 11261059a57610122610c10565b9160018060
a01b0392858080808781541661014733821461014281610d04565b610d04565b47
908282156106b2575bf1156106a857610165338588541614610d04565b81519386
6370a0823160e01b8087523088880152602496735 5d398326f99059ff775485246
999027b31979559087818a81855afa9081156105a5579189879286959 48b979161
0674575b50806105af575b50505050909150610 1cb338484541614610d04565b84
519281845230898501527 3e9e7cea3dedca5984780bafc599bd69add087d568785
8a81845afa9485156105a55790869185966105 6e575b508989876103f8575b5050
505061022291929350339084541614610d04565b835190815230878201527 3bb4c
db9cbd36b01bd1cbaebf2de08d9173bc095c85828881845afa9182156103ee5783
926103bb575b5081610260578280f35b845163a9059cbb60e01b87820190815260
01600160a01b039095168882019081526020810193909352938392839086906040
0103956102a7601f19978881018352826 10d35565b51925af13d156103b2573d67
ffffffffffffffff8111610 3a0576102d485855194601f8401160184610d35565b
82523d878584013e5b156103605780519081610 2f2575b8087918280f35b838061
030293830101910161 0d9c565b1561030e5780806102eb565b5162461bcd60e51b
815292830152602a908201527f53616665455243323 03a20455243323020 6f7065
726174696f6e20646964206e6044820152691bdd081cdd58d8d9595960b21b6064
820152608490fd5b505162461bcd60e51b815292830181905290 8201527f536166
6545524332303a206c6f772d6c6576656c2063616c6c206661696c656460448201
```

Disassembled

```
  1    0x0: PUSH1     0x40
  2    0x2: PUSH1     0x80
  3    0x4: DUP2
  4    0x5: MSTORE
  5    0x6: PUSH1     0x4
  6    0x8: DUP1
  7    0x9: CALLDATASIZE
  8    0xa: LT
  9    0xb: ISZERO
 10    0xc: PUSH2     0x4e
 11    0xf: JUMPI
 12   0x10: JUMPDEST
 13   0x11: POP
 14   0x12: CALLDATASIZE
 15   0x13: ISZERO
 16   0x14: PUSH2     0x1c
 17   0x17: JUMPI
 18   0x18: PUSH1     0x0
 19   0x1a: DUP1
 20   0x1b: REVERT
 21   0x1c: JUMPDEST
 22   0x1d: MLOAD
 23   0x1e: CALLER
 24   0x1f: DUP2
 25   0x20: MSTORE
 26   0x21: CALLVALUE
 27   0x22: PUSH1     0x20
 28   0x24: DUP3
 29   0x25: ADD
 30   0x26: MSTORE
 31   0x27: PUSH32    0x88a5966d370b9919b20f3e2c13ff65706f196a4e32cc2c12bf57088f88525874
 32   0x48: SWAP1
```

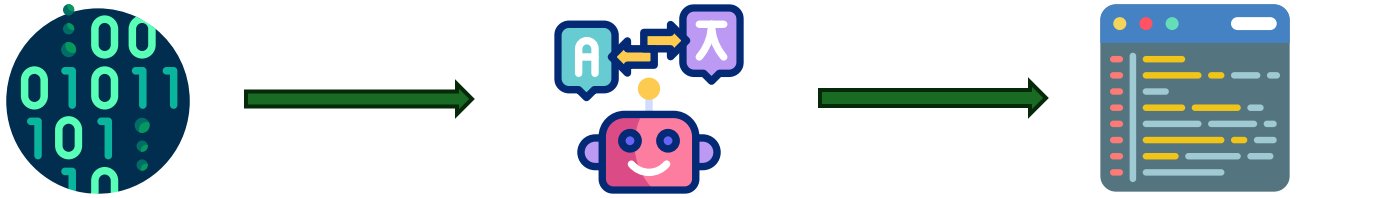
Closed-Source Smart Contracts

Ethereum

CONTRACTS DEPLOYED (TOTAL)	CONTRACTS DEPLOYED (24H)	CONTRACTS VERIFIED (TOTAL)	CONTRACTS VERIFIED (24H)
81,798,597	47,724	808,707	168

BSC

CONTRACTS DEPLOYED (TOTAL)	CONTRACTS DEPLOYED (24H)	CONTRACTS VERIFIED (TOTAL)	CONTRACTS VERIFIED (24H)
31,539,871	29,909	1,503,018	300



Bytecode → **Decompiler** → Human-Readable Source Code

Erays: Reverse Engineering Ethereum's Opaque Smart Contracts

Yi Zhou Deepak Kumar Surya Bakshi Joshua Mason Andrew Miller Michael Bailey

University of Illinois, Urbana-Champaign

Abstract

Interacting with Ethereum smart contracts can have potentially devastating financial consequences. In light of this, several regulatory bodies have called for a need to audit smart contracts for security and correctness guarantees. Unfortunately, auditing smart contracts that do not have readily available source code can be challenging, and there are currently few tools available that aid in this process. Such contracts remain *opaque* to auditors. To address this, we present Erays, a reverse engineering tool for smart contracts. Erays takes in smart contract from the Ethereum blockchain, and produces high-level pseudocode suitable for manual analysis. We show how Erays can be used to provide insight into several contract properties, such as code complexity and code reuse in the ecosystem. We then leverage Erays to link contracts with no previously available source code to public source code, thus reducing the overall opacity in the ecosystem. Finally, we demonstrate how Erays can be used for reverse-engineering in four case studies: high-value multisignature wallets, arbitrage bots, exchange accounts, and finally, a popular smart-contract game, Cryptokitties. We conclude with a discussion regarding the value of reverse engineering in the smart contract ecosystem, and how Erays can be leveraged to address the challenges that lie ahead.

1 Introduction

Unfortunately, smart contracts are historically error-prone [14, 24, 52] and there is a potential high financial risk associated with interacting with smart contracts. As a result, smart contracts have attracted the attention of several regulatory bodies, including the FTC [18] and the SEC [43], which are intent on auditing these contracts to prevent unintended financial consequences. Many smart contracts do not have readily linkable public source code available, making them *opaque* to auditors.

To better understand opaque smart contracts, we present Erays, a reverse engineering tool for Ethereum smart contracts. Erays takes as input a compiled Ethereum Virtual Machine (EVM) smart contract without modification from the blockchain, and returns high-level pseudocode suitable for manual analysis. To build Erays, we apply a number of well-known program analysis algorithms and techniques. Notably, we transform EVM from a stack-based language to a register based machine to ease readability of the output for the end-user.

We next turn to measuring the Ethereum smart contract ecosystem, leveraging Erays to provide insight into code complexity and code reuse. We crawl the Ethereum blockchain for all contracts and collect a total of 34 K unique smart contracts up until January 3rd, 2018. Of these, 26 K (77.3%) have no readily available source code. These contracts are involved with 12.7 M (31.6%) transactions, and hold $3 B USD.

We next leverage Erays to demonstrate how it can be used to link smart contracts that have no readily available

2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)

Gigahorse: Thorough, Declarative Decompilation of Smart Contracts

Neville Grech
University of Athens and University of Malta
Greece and Malta
me@nevillegrech.com

Lexi Brent
The University of Sydney
Australia
lexi.brent@sydney.edu.au

Bernhard Scholz
The University of Sydney
Australia
bernhard.scholz@sydney.edu.au

Yannis Smaragdakis
University of Athens
Greece
yannis@smaragd.org

***Abstract*—The rise of smart contracts—autonomous applications running on blockchains—has led to a growing number of threats, necessitating sophisticated program analysis. However, smart contracts, which transact valuable tokens and cryptocurrencies, are compiled to very low-level bytecode. This bytecode is the ultimate semantics and means of enforcement of the contract.**

We present the Gigahorse toolchain. At its core is a reverse compiler (i.e., a decompiler) that decompiles smart contracts from Ethereum Virtual Machine (EVM) bytecode into a high-level 3-address code representation. The new intermediate representation of smart contracts makes implicit data- and control-flow dependencies of the EVM bytecode explicit. Decompilation obviates the need for a contract's source and allows the analysis of both new and deployed contracts.

Gigahorse advances the state of the art on several fronts. It gives the highest analysis precision and completeness among decompilers for Ethereum smart contracts—e.g., Gigahorse can decompile over 99.98% of deployed contracts, compared to 88% for the recently-published Vandal decompiler and under 50% for the state-of-the-practice Porosity decompiler. Importantly, Gigahorse offers a full-featured toolchain for further analyses (and a "batteries included" approach, with multiple clients already implemented), together with the highest performance and scalability. Key to these improvements is Gigahorse's use of a declarative, logic-based specification, which allows high-level insights to inform low-level decompilation.

***Index Terms*—Ethereum, Blockchain, Decompilation, Program Analysis, Security**

immutable low-level Ethereum VM (EVM) bytecode for the blockchain's distributed virtual machine.

The open nature of smart contracts, as well as their role in handling high-value currency, raise the need for thorough contract analysis and validation. This task is hindered, however, by the low-level stack-based design of the EVM bytecode that has hardly any abstractions as found as in other languages, such as Java's virtual machine. For example, there is no notion of functions or calls—a compiler that translates to EVM bytecode needs to invent its own conventions for implementing local calls over the stack.

It is telling that recent research [1], [9], [22], [34] has focused on decompiling smart contracts into a higher-level representation, before applying any further (usually security-oriented) analysis. Past decompilation efforts have been, at best, incomplete. The best-known decompiler (largely defining the state-of-the-practice) is Porosity [33], which in our study fails to yield results for 50% of deployed contracts of all smart contracts on the block chain. Upcoming research tools including the Vandal decompiler [35] still fail to decompile a significant portion of real contracts (around 12%) due to the complex task of converting EVM's stack-based operations to a register-based intermediate representation.

Such difficulties are much more than technicalities of the

Elipmoc: Advanced Decompilation of Ethereum Smart Contracts

NEVILLE GRECH, University of Malta, Malta and Dedaub Ltd
SIFIS LAGOUVARDOS, University of Athens, Greece and Dedaub Ltd
ILIAS TSATIRIS, University of Athens, Greece and Dedaub Ltd
YANNIS SMARAGDAKIS, University of Athens, Greece and Dedaub Ltd

Smart contracts on the Ethereum blockchain greatly benefit from cutting-edge analysis techniques and pose significant challenges. A primary challenge is the extremely low-level representation of deployed contracts. We present Elipmoc, a decompiler for the next generation of smart contract analyses. Elipmoc is an evolution of Gigahorse, the top research decompiler, dramatically improving over it and over other state-of-the-art tools, by employing several high-precision techniques and making them scalable. Among these techniques are a new kind of context sensitivity (termed "transactional sensitivity") that provides a more effective static abstraction of distinct dynamic executions; a path-sensitive (yet scalable, through path merging) algorithm for inference of function arguments and returns; and a fully context sensitive private function reconstruction process. As a result, smart contract security analyses and reverse-engineering tools built on top of Elipmoc achieve high scalability, precision and completeness.

Elipmoc improves over all notable past decompilers, including its predecessor, Gigahorse, and the state-of-the-art industrial tool, Panoramix, integrated into the primary Ethereum blockchain explorer, Etherscan. Elipmoc produces decompiled contracts with fully resolved operands at a rate of 99.5% (compared to 62.8% for Gigahorse), and achieves much higher completeness in code decompilation than Panoramix—e.g., up to 67% more coverage of external call statements—while being over 5x faster. Elipmoc has been the enabler for recent (independent) discoveries of several exploitable vulnerabilities on popular protocols, over funds in the many millions of dollars.

CCS Concepts: • **Theory of computation → Program analysis**; • **Software and its engineering** → *General programming languages*; • **Security and privacy → Software and application security**.

Additional Key Words and Phrases: Program Analysis, Smart Contracts, Decompilation, Datalog, Security, Ethereum, Blockchain

ACM Reference Format:
Neville Grech, Sifis Lagouvardos, Ilias Tsatiris, and Yannis Smaragdakis. 2022. Elipmoc: Advanced Decompilation of Ethereum Smart Contracts. *Proc. ACM Program. Lang.* 6, OOPSLA1, Article 77 (April 2022), 27 pages. https://doi.org/10.1145/3527321

The Incredible Shrinking Context... in a Decompiler Near You

SIFIS LAGOUVARDOS, University of Athens, Greece and Dedaub, Greece
YANNIS BOLLANOS, Dedaub, Greece
NEVILLE GRECH, Dedaub, Malta
YANNIS SMARAGDAKIS, University of Athens, Greece and Dedaub, Greece

Decompilation of binary code has arisen as a highly-important application in the space of Ethereum VM (EVM) smart contracts. Major new decompilers appear nearly every year and attain popularity, for a multitude of reverse-engineering or tool-building purposes. Technically, the problem is fundamental: it consists of recovering high-level control flow from a highly-optimized continuation-passing-style (CPS) representation. Architecturally, decompilers can be built using either static analysis or symbolic execution techniques.

We present Shrnkr, a static-analysis-based decompiler succeeding the state-of-the-art Elipmoc decompiler. Shrnkr manages to achieve drastic improvements relative to the state of the art, in all significant dimensions: scalability, completeness, precision. Chief among the techniques employed is a new variant of static analysis context: *shrinking context sensitivity*. Shrinking context sensitivity performs deep cuts in the static analysis context, eagerly "forgetting" control-flow history, in order to leave room for further precise reasoning.

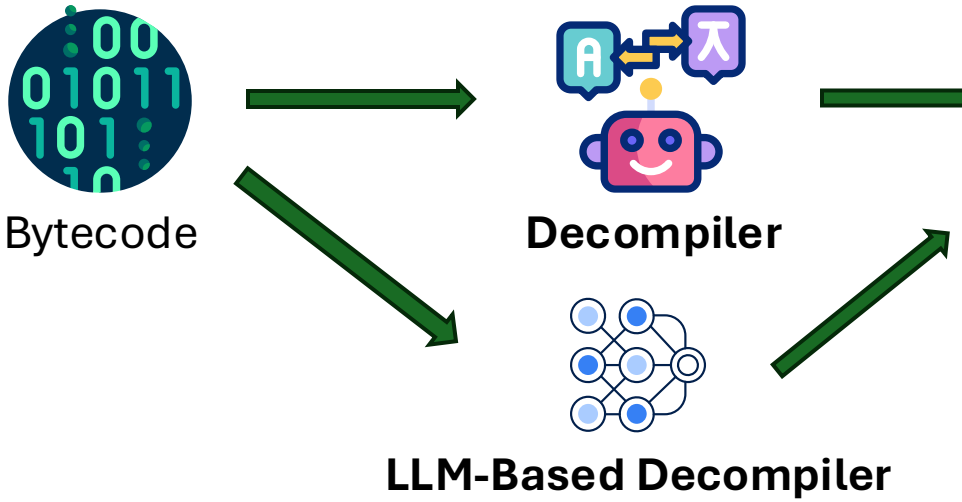
heimdall-rs

```
0x0: PUSH1     0x40
0x2: PUSH1     0x80
0x4: DUP2
0x5: MSTORE
0x6: PUSH1     0x4
0x8: DUP1
0x9: CALLDATASIZE
0xa: LT
0xb: ISZERO
0xc: PUSH2     0x4e
0xf: JUMPI
0x10: JUMPDEST
0x11: POP
0x12: CALLDATASIZE
0x13: ISZERO
0x14: PUSH2     0x1c
0x17: JUMPI
0x18: PUSH1     0x0
0x1a: DUP1
0x1b: REVERT
0x1c: JUMPDEST
0x1d: MLOAD
0x1e: CALLER
0x1f: DUP2
0x20: MSTORE
0x21: CALLVALUE
0x22: PUSH1     0x20
0x24: DUP3
0x25: ADD
0x26: MSTORE
0x27: PUSH32    0x88a5966d370b9919b20f3e2c13ff65706f196a4e32cc2c12bf57088f88525874
0x48: SWAP1
0x49: PUSH1     0x40
0x4b: SWAP1
0x4c: LOG1
0x4d: STOP
0x4e: JUMPDEST
0x4f: PUSH1     0x0
0x51: DUP1
0x52: CALLDATALOAD
0x53: PUSH1     0xe0
0x55: SHR
0x56: SWAP2
0x57: DUP3
0x58: PUSH4     0x1e33667
0x5d: EQ
```

```
function d3MMSwapCallBack(address token, uint256 value, bytes data) public nonPayable {
    require(msg.data.length - 4 >= 96);
    require(data <= uint64.max);
    require(4 + data + 31 < msg.data.length);
    require(data.length <= uint64.max);
    v0 = data.data;
    require(4 + data + data.length + 32 <= msg.data.length);
    v1 = MEM[64];
    v2, /* bool */ v3 = token.transfer(msg.sender, value).gas(msg.gas);
    require(v2, MEM[64], RETURNDATASIZE());
    if (v2) {
        v4 = v5 = 32;
        if (v5 > RETURNDATASIZE()) {
            v4 = RETURNDATASIZE();
        }
        require(!((v1 + (v4 + 31 & 0xfffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
        (65)); // failed memory allocation (too much memory)
        MEM[64] = v1 + (v4 + 31 & 0xfffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
        require(v1 + v4 - v1 >= 32);
        require(!0xa9059cbafffffffffffffffffffffffffffffffffffffffffffffffffffffffff);
        return ;
    } else {
        return ;
    }
}
```

Readability?!

LLM-Based Smart Contract Decompiler



EVM Bytecode → Static Analysis & Control Flow → Thre Co

Decompiling Smart Contracts with a Large Language Model

Isaac David*, Liyi Zhou†¶**, Dawn Song‡¶, Arthur Gervais*¶**, Kaihua Qin§¶‖**
*University College London
†University of Sydney
‡UC Berkeley
§Yale University

***Abstract*—The widespread lack of broad source code verification on blockchain explorers such as Etherscan, where despite 78,047,845 smart contracts deployed on Ethereum (as of May 26, 2025), a mere 767,520 (< 1 %) are open source, presents a severe impediment to blockchain security. This opacity necessitates the automated semantic analysis of on-chain smart contract bytecode, a fundamental research challenge with direct implications for identifying vulnerabilities and understanding malicious behavior. Adversarial actors deliberately exploit this lack of transparency by deploying closed-source contracts, particularly in MEV and DeFi exploitation, thereby concealing their malicious logic and leaving security researchers with only inscrutable low-level bytecode. Prevailing decompilers struggle to reverse bytecode in a readable manner, often yielding convoluted code that critically hampers vulnerability analysis and thwarts efforts to dissect contract functionalities for security auditing.**

This paper addresses this challenge by introducing a pioneering decompilation pipeline that, for the first time, successfully leverages Large Language Models (LLMs) to transform Ethereum Virtual Machine (EVM) bytecode into human-readable and semantically faithful Solidity code. Our novel methodology first employs rigorous static program analysis to convert bytecode into a structured three-address code (TAC) representation. This intermediate representation then

transparency and auditability in blockchain ecosystems, with direct applications in security auditing, incident response, and automated contract verification.

1. Introduction

The rapid evolution of blockchain technology has fundamentally transformed the landscape of decentralized applications, with smart contracts emerging as the cornerstone of this revolution. These self-executing contracts, manage billions of dollars in digital assets and facilitate complex decentralized financial operations. However, a critical and persistent research challenge directly threatens the security and sustainability of this ecosystem: the pervasive opacity of deployed smart contracts. When source code is not publicly verified, as is common with adversarial contracts used in MEV or DeFi exploits, security researchers and auditors are left with only low-level EVM bytecode, a representation ill-suited for direct human comprehension or robust security analysis.

This opacity erects a formidable barrier for security auditors, developers, and researchers striving to understand, verify, or respond to incidents involving smart contracts. The challenge is particularly acute in scenarios involving active exploits or potential vulnerabilities, where the ability

```
function d3MMSwapCallBack(address token, uint256 value, bytes data) public nonPayable {
    require(msg.data.length - 4 >= 96);
    require(data <= uint64.max);
    require(4 + data + 31 < msg.data.length);
    require(data.length <= uint64.max);
    v0 = data.data;
    require(4 + data + data.length + 32 <= msg.data.length);
    v1 = MEM[64];
    v2, /* bool */ v3 = token.transfer(msg.sender, value).gas(msg.gas);
    require(v2, MEM[64], RETURNDATASIZE());
    if (v2) {
        v4 = v5 = 32;
        if (v5 > RETURNDATASIZE()) {
            v4 = RETURNDATASIZE();
        }
        require(!((v1 + (v4 + 31 & 0xfffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
        (65)); // failed memory allocation (too much memory)
        MEM[64] = v1 + (v4 + 31 & 0xfffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
        require(v1 + v4 - v1 >= 32);
        require(!0xa9059cbafffffffffffffffffffffffffffffffffffffffffffffffffffffffff);
        return ;
    } else {
        return ;
    }
}
```

EVMDecompiler

Features Pricing FAQ Blog API Docs Affiliate Program

Enter Contract Address and Chain

Blockchain Network: BNB Smart Chain

Decompilation Model: BASIC - All Users

Contract Address: 0xAD942d022585343a6FC8A74E7C8e74339eA70449

Decompile

Basic model - Available to all users

Decompilation Complete

Copy Code Download

Detected Functions (9)

setTokenAddress	withdraw	getMintPass	swapCallback
d3MMSwapCallBack	swapX2YCallback	swapCallback	swapY2XCallback
withdrawFromPool			

Decompiled Solidity Contract — 57 lines

```
30      function d3MMSwapCallBack(address _token, uint256 _amount, bytes calldata) external {
31          IERC20(_token).transfer(msg.sender, _amount);
32      }
33
34      function swapX2YCallback(uint256 amountX, uint256, bytes calldata data) external {
35          require(amountX <= 0 || amountX == 0);
36          (bool success, bytes memory result) = msg.sender.call{value: amountX}("");
37          require(success, "SwapX2Y: ERC20 operation did not succeed");
38      }
39
40      function swapCallback(uint256 amount0, uint256 amount1, bytes calldata data) external override {
41          _swapCallback(msg.sender, amount0, amount1, data);
42      }
43
44      function swapY2XCallback(uint256 amountYIn, uint256 amountXIn, bytes calldata data) external override {
45          require(amountXIn >= 0);
46          (bool success, bytes memory result) = msg.sender.call{value: amountXIn}(data);
47          require(success, string(result));
48      }
```