SMARTMARK: Software Watermarking Scheme for Smart Contracts

***Abstract*—A smart contract is a self-executing program on a blockchain to ensure an immutable and transparent agreement without the involvement of intermediaries. Despite its growing popularity for many blockchain platforms like Ethereum, no technical means is available even when a smart contract requires to be protected from being copied. One of promising directions to claim a software ownership is software watermarking. However, applying existing software watermarking techniques is challenging because of the unique properties of a smart contract, such as a code size constraint, non-free execution cost, and no support for dynamic allocation under a virtual machine environment. This paper introduces a novel software watermarking scheme, dubbed SMARTMARK, aiming to protect the ownership of a smart contract against a pirate activity. SMARTMARK builds the control flow graph of a target contract runtime bytecode, and locates a collection of bytes that are randomly elected for representing a watermark. We implement a full-fledged prototype for Ethereum, applying SMARTMARK to 27,824 unique smart contract bytecodes. Our empirical results demonstrate that SMARTMARK can effectively embed a watermark into a smart contract and verify its presence, meeting the requirements of credibility and imperceptibility while incurring an acceptable performance degradation. Besides, our security analysis shows that SMARTMARK is resilient against viable watermarking corruption attacks; *e.g.*, a large number of dummy opcodes are needed to disable a watermark effectively, resulting in producing an illegitimate smart contract clone that is not economical.**

I. INTRODUCTION

Due to the advancements in blockchain technologies, blockchain-based smart contracts (hereinafter referred to as *smart contracts*) have received significant attention from both academia and industry over the last few years. A vast number of smart contracts have already been deployed on blockchains (*e.g.*, over 10 millions in 2020 on the Ethereum network [1]).

As a smart contract has been adopted for business, we encounter a new (but familiar) challenge on protecting it when a smart contract owner needs to claim one's intellectual property right. Since a smart contract is a type of programming code, it is inherently prone to be plagiarized. Although a smart contract is deployed in a binary form on a blockchain like other software distributions, its size constraint (*e.g.*, 24KB for Ethereum) makes a reverse engineering relatively less painful with the state-of-the-art tools (*e.g.*, Erays [2], Vandal [3], Gigahorse [4]). Recent studies [5], [6] reveal that a vast amount of contract code blocks had been indeed cloned. He et al. [1] identified 41 decentralized applications (DApps[1]) with 73 plagiarized DApps, which may cause a substantial financial loss (over USD 19 million on Sep 21, 2019) to the original DApp creators. Besides, they showed that careless code reuse could bring about an unwanted result from a security perspective.

[1]A DApp is a collection of smart contracts incorporated with an interface on a website or an application, to interact with users.

A well-known technique for protecting a software copyright is software watermarking, a process of embedding a watermark W into a program P such that W can be further detected or extracted to assert the ownership of P [7]. In essence, the underlying mechanism is that W would be present when one copies P even with the attempt of a corruption (*e.g.*, modification). A plethora of software watermarking schemes [8], [9], [10], [11], [7], [12], [13], [14], [15], [16], [17], [18], [19], [20], [21], [22] have been introduced against a piracy.

However, applying the existing techniques to a smart contract is not viable due to its unique properties. First, hiding a watermark within a program would be difficult because the size of a smart contract is typically smaller than that of a conventional program. Consequently, it is possible for a skillful adversary to manipulate the watermark with static code analysis. Second, a virtual machine such as Ethereum Virtual Machine (EVM) offers a different enviornment from a bare machine, making it infeasible to adopt prior approches like the return-oriented programming (ROP)-based scheme [22] or the function reodering scheme [11]. Third, running a bytecode under EVM inevitably incurs a transaction cost, which restricts any scheme that introduces additional code or data. Fourth, EVM does not have a feature of dynamic memory allocation, which disables the adoption of a prior scheme [23], [7]) to utilize that feature.

Only a few attempts have been made to protect smart contract developers' intellectual property rights. Zhang et al. [24] present code obfuscation techniques for a smart contract, especially written in Solidity [25], which makes it difficult to decompile a smart contract. Yan et al. [26] propose a technique to increase the difficulty level of recovering a control flow graph from a smart contract bytecode by introducing four anti-reverse engineering code patterns. Although it is possible to raise the bar with the above techniques, they are still far from a comprehensive solution for verifying the originality of a smart contract (*i.e.*, Have a contract been copied (partly or in full) from another?).

In this work, we present SMARTMARK, to the best of our knowledge, the first watermarking scheme on smart contracts that considers both varying requirements of a watermark (*i.e.*, imperceptibility, spread, credibility, resiliency, capacity, efficiency) and unique properties of smart contracts (*i.e.*, gas cost). At a high level, SMARTMARK builds a control flow graph (CFG) from the runtime bytecode of a smart contract and randomly elects a series of bytes from the blocks selected across the CFG as a watermark.[2] Next, SmartMark creates a data structure that holds essential information (*e.g.*, the locations of elected bytes and the CFG generation method) to extract the watermark later. SmartMark privately keeps this data structure and only inserts the hash of the data structure into the creation bytecode of the smart contract for further verification of the watermark to claim the originality of the smart contract. Note that we assume that SmartMark does not add any extra code for watermarking to the runtime bytecode of the smart contract, indicating that the smart contract's functional behavior would remain the same without incurring additional gas costs to execute the smart contract.

The main benefits of our design choice are free from ① an undesirable gas cost for a transaction as the size of a runtime bytecode stays intact, and ② detection techniques based on a static analysis as no additional code is introduced for a watermark. To this end, we develop a full-fledged prototype of SmartMark and demonstrate how our approach fulfills the requirements of a watermarking scheme on smart contracts at an acceptable cost. Moreover, we collected all the blockchain blocks (about nine million blocks between 30 July 2015 and 21 June 2022) from the Ethereum Mainnet, and selected 27,824 unique bytecodes from those blocks to evaluate the effectiveness and efficiency of SmartMark.

The contribution of our paper is summarized as follows.

- We present SmartMark, a novel software watermarking scheme that satisfies varying requirements of watermarking for smart contracts.
- We empirically evaluate SmartMark, demonstrating its effectiveness and efficiency.
- We thoroughly study the resiliency of our watermarking scheme against viable attacks.

II. Background

This section describes the background of the Ethereum smart contract and the virtual machine environment to run it.

A. Smart Contract and License

Smart Contract. The term "smart contract" [27] refers to a piece of programming code that is permanently stored and executed for processing transactions when predetermined conditions are met. Smart contracts act as nodes or accounts on the blockchain by leveraging its tamper-resiliency, traceability, and transparency. Ethereum [28] is one of the most popular and prominent blockchain-based smart contract platforms. The smart contracts on Ethereum are written in a high-level programming language such as Solidity. A Solidity code must be compiled into Ethereum bytecode [28] to properly run on a blockchain, remaining immutable and indelible within a blockchain ledger. We design SmartMark by embedding a watermark into bytecodes (compiled from a smart contract written in Solidity at the Ethereum platform) and extracting it from the bytecodes. Therefore, SmartMark is agnostic to specific features of a high-level programming language.

License in Solidity. The Solidity compiler offers the means of a machine-readable SPDX license identifier [29] by default, which can be embedded into a bytecode as metadata (by inserting a specific license header into every source file). However, the license identifier differs from a digital watermark because it merely represents one of the standard licenses (*e.g.*, MIT, Apache, BSD, Creative Commons) rather than specifying the actual ownership of a smart contract, being inappropriate for claiming a smart contract copyright.

B. EVM and Bytecode

Ethereum Virtual Machine (EVM). The Ethereum Virtual Machine (EVM) offers a stack-based runtime environment for smart contracts, where a chunk of bytecodes can be executed upon receiving a transaction. EVM maintains varying machine states that hold a data structure as an execution component, including accounts, balances, stack, memory, storage, and a program counter. EVM supports 150 instructions [30] where each comprises a single byte opcode (mnemonic) and zero or more operands.

Gas Cost. Ethereum introduces the notion of the execution fee, dubbed *gas*, for every EVM opcode (*e.g.*, the opcode for multiplication consumes five units of gas) based on the computational and storage overheads of each opcode [30]. This enables miners to obtain a reward for computational resources to store smart contracts and execute them. Besides, it can prevent denial-of-service (DoS) attacks that invoke a time-consuming function [31].

Bytecode for EVM. Solidity emits two types of bytecode: ① creation bytecode (*i.e.*, *init* bytecode) for initializing (*i.e.*, constructor) and deploying a contract, and ② runtime bytecode (*i.e.*, *deployment* bytecode) for executing the contract that is stored on a blockchain. The major difference in spending a cost is that a creation bytecode requires a gas once whereas a runtime bytecode consumes a gas in every transaction. Note that one can obtain runtime bytecodes on a blockchain and creation bytecode from a contract transaction log.

III. Smart Contract Watermarking

This section describes the need for a watermark technique on smart contracts, challenges, varying requirements, and a threat model with viable attacks.

A. Motivation and Challenges

Motivation. Due to the nature of public blockchain platforms, even if the smart contract authors do not make source code publicly available, smart contracts can be exposed in a bytecode format and could be reused by anyone. Indeed, recent empirical studies [5], [6] reveal that code reuse in smart contracts is quite prevalent. For example, Chen et al. [6] discovered that 26% of contract code blocks had been cloned (14.6 occurrences on average) from the $146K$ open-sourced projects, suggesting common patterns of code duplication in

[2]A runtime bytecode is the execution body of a smart contract, and a creation bytecode consists of both a runtime and initialization bytecode (*i.e.*, constructor).

smart contracts. Although the power of reusability helps a smart-contracts-driven ecosystem to be rich, it may pose a severe threat to managing smart contracts' intellectual property rights (IPR). He et al. [1] demonstrated that over 96% of 10 million contracts had duplicates. It also showed that there were 73 plagiarized DApps, estimating substantial financial losses (over USD 19 million on Sep 21, 2019) for the original DApp creators.[3] Such a growing concern motivates our work for protecting the IPR of a smart contract. However, it is non-trivial to prevent reusing existing smart contracts. A possible solution would be to develop a software watermarking [7] scheme to provide a technical means that claims the originality of a smart contract on demand.

Challenges. Applying prior software watermarking techniques to a smart contract is challenging due to the characteristics of a smart contract programming language. One of the biggest hurdles is that smart contracts typically have a small size (up to 24KB), making it difficult to conceal a watermark from the original smart contract code. Another challenge is that running a bytecode under EVM comes with a (gas) cost (*i.e.*, Ethereum transaction fee), possibly leading to avoiding any watermarking technique unless it stays the total cost intact. Hence, it is evident that a watermarking scheme with a charge would not be welcomed even with the presence of the technique. Lastly, the EVM environment for a smart contract does not allow for dynamically allocated memory, disabling the adoption of existing watermarking schemes [23], [7] that utilize dynamic allocation.

Goal. By nature, a blockchain is designed to confirm if a certain smart contract appears for the first time. However, a technical means is absent to verify that a suspicious smart contract is a replica of an existing smart contract (thereby violating IPR). In this paper, we aim to provide such a means to address this problem. To the best of our knowledge, we introduce the first watermarking scheme for smart contracts that should be able to tackle the aforementioned challenges.

B. Requirements

Instead of reinventing the wheel for the requirements of a watermarking scheme, we adopt the general ones as with previous software watermark approaches [11], [7], [32], [33], [34], [35], [14]: imperceptibility, spread, credibility, resiliency, capacity, and efficiency. Besides, we define a cost (gas consumption) as another requirement for contract watermarking.

- *Imperceptibility* ensures that a watermark must be scarcely perceptible, that is, a smart contract with an embedded watermark must be indistinguishable from the one without.
- *Spread* denotes how well a watermark is distributed across the whole smart contract code. Typically, a well-scattered watermark tends to be resilient against corruption attempts.
- *Credibility* ensures that a watermark must be reliably verifiable, minimizing false positive or negative cases.

[3]The plagiarized DApps collected 89,565.32 ETH (around 30% of the market), becoming a loss of the 41 benign DApps.

- *Resiliency* represents the robustness of a watermarking scheme against tampering attacks that aim to invalidate a watermark, including addition, subtraction, and distortion.
- *Capacity* represents the data rate of a watermark that can be encoded into a target contract. Considering a contract size constraint, the length of a watermark cannot exceed it.
- *Efficiency* represents a performance overhead (*i.e.*, computational resource) that is needed for watermarking operations. We separately define a gas cost metric for smart contracts.
- *Cost* represents the amount of gas consumption for inserting and validating a watermark. We utilize a gas price per individual opcode as pre-defined in [30].

C. Threat Model

The objective of our watermarking scheme for a smart contract is to thwart an adversary's considerable efforts with reasonable resources rather than suggesting an unbreakable scheme (as a fully motivated attacker could hardly be prevented). With this in mind, in this paper, we assume a strong adversary who is capable of ① either possessing source code or solely obtaining an open bytecode (as a more general case) for a given smart contract, ② understanding our watermark scheme beforehand, and ③ performing arbitrary code manipulation on source code (Section VI-G) or bytecode, attempting to tamper with a watermark. We also assume that the adversary could collect as many smart contracts as possible for further comparisons between them (*i.e.*, collusive attack).

Attack Types. We classify five viable attacks largely into two categories: passive attacks (denoted as "P"), including unauthorized recognition and collusion (against imperceptibility), and active attacks (denoted as "A"), including addition, deletion, and distortion (against credibility and resiliency). Our design principle does not necessarily conceal the presence of a watermark embedded in smart contracts at all. Hence, recognizing such information itself would not weaken the overall security of SMARTMARK unless an adversary could reveal the exact location of a watermark.

- *(P) Unauthorized recognition* refers to an attack that identifies the location of a watermark in a target smart contract.
- *(P) Collusion* refers to an attack that recognizes the location of a watermark in a target contract by comparison.
- *(A) Addition* refers to an attack that embeds another watermark (*i.e.*, adversary's ownership) into a target contract.
- *(A) Deletion* refers to an attack that eliminates a valid watermark from a target smart contract.
- *(A) Distortion* refers to an attack that encompasses every transformation for damaging an existing watermark.

IV. SMARTMARK DESIGN

This section sketches the design of SMARTMARK that considers both the requirements for generic watermarks and the smart contract's idiosyncratic properties. Notably, our scheme does not introduce additional runtime bytecode because implanting bytes would be revealed by a sophisticated attack as well as increasing an undesirable gas cost.

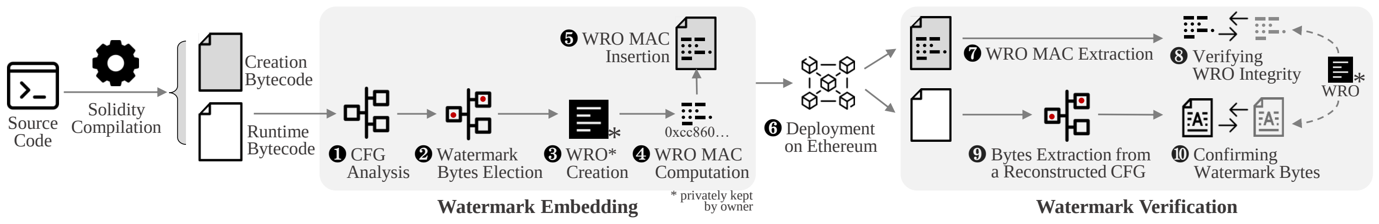


Fig. 1. Overall SMARTMARK scheme for embedding and verifying a watermark for a smart contract. We carefully elect bytes (❷) that comprises a watermark from the CFG (❶) of a runtime bytecode, creating a watermark reference object (WRO) (❸). The original author secretly holds the WRO, and computes its hash (WRO MAC). The WRO MAC is computed and embedded to a creation bytecode (❹, ❺), followed by deploying it on the Ethereum network (❻). By verifying the extracted WRO MAC (❼, ❽) from a creation bytecode with the extracted bytes from a reconstructed CFG (❾), a watermark can be verified (❿) on demand.

A. Design Overview

Fig. 1 depicts the overall process of SMARTMARK for embedding and verifying a watermark for a smart contract. First, the Solidity compiler generates both creation bytecodes (*i.e.*, contract constructor) and runtime bytecodes (*i.e.*, actual execution code under EVM). For embedding a watermark, our scheme harnesses both creation and runtime bytecodes for watermarking over smart contracts. We first construct a CFG from a runtime bytecode (❶ in Fig. 1), followed by electing a series of bytes that incorporates a watermark (❷). Then, we explicitly define a structure (dubbed *watermark reference object*; WRO) that contains essential information to represent the watermark, which must be privately maintained by the owner of a smart contract (❸). We compute the hash value of the WRO (❹) and insert it into a creation bytecode (❺) for further validating a watermark. For brevity, hereinafter, we call such a hash value a *WRO MAC* that represents a message authentication code for WRO. Once the creation bytecode is complete, it is deployed to the Ethereum network (❻). It is worth noting that all bytecodes are publicly available in a tamper-proof fashion after deployment, which indicates that a WRO MAC is publicly accessible and immutable. To verify the presence of a watermark in a target smart contract, a verifier (e.g., the owner of the original smart contract) uses the WRO containing the information about the watermark. For the validity of the WRO, the verifier first computes the hash value of the WRO and compares it with the WRO MAC extracted from a creation bytecode of the target contract (❼, ❽). If the two values are matched, the WRO is valid. Therefore, the verifier reconstructs a CFG from a target smart contract (❾), and confirms the presence of a watermark in the CFG with the information in the WRO (❿).

B. Watermark Embedding for Smart Contracts

This section portrays SMARTMARK's watermark embedding for smart contracts under the hood.

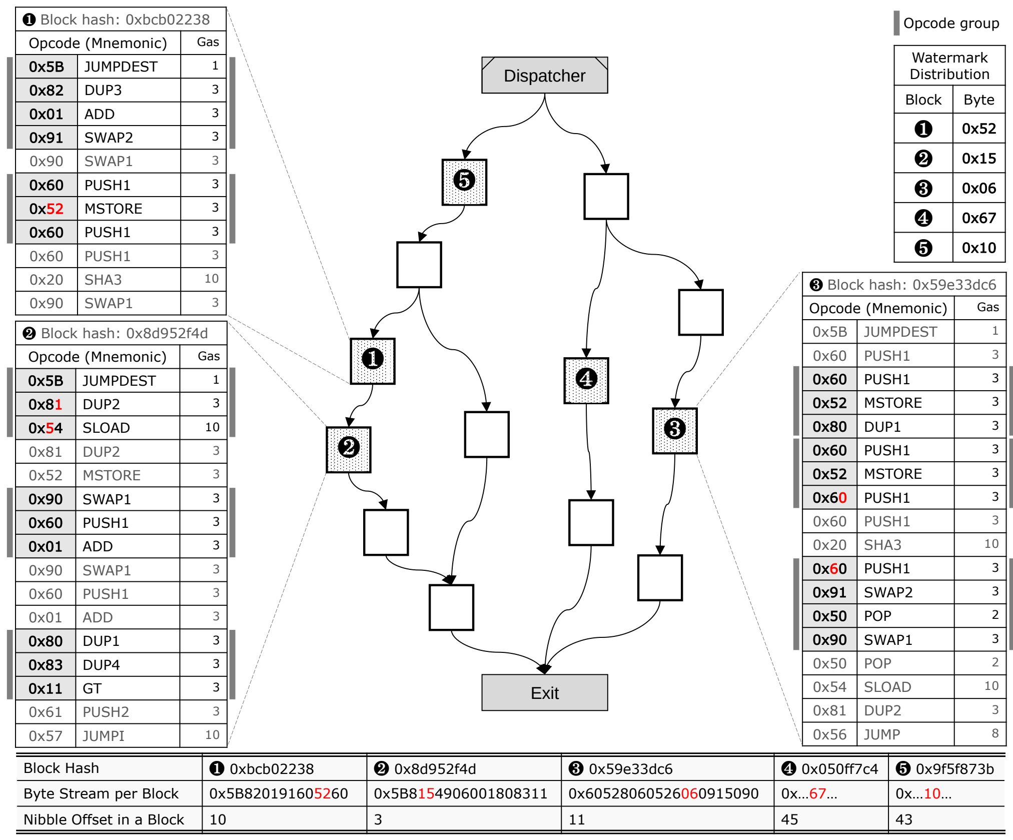
1) Design Choice: SMARTMARK carefully considers the two types of bytecode with different properties; a runtime bytecode for electing the bytes that form a watermark and a creation bytecode for ensuring the integrity of a WRO. SMARTMARK does not introduce any extra code within a runtime bytecode since the execution of a smart contract inevitably requires a gas cost. Alternatively, we store a 32-byte WRO MAC as a variable in the creation bytecode because it only incurs a one-time cost during deployment. Besides, we embed multiple (N) watermarks, enabling a robust recovery in case of partial damage (Section IV-C).

2) Strategic Byte Election: As illustrated in Fig. 1, we begin with constructing a CFG from a runtime bytecode. However, not every byte accounts for a watermark in SMARTMARK because strategic insertion of a (cheap) dummy byte into every block within a target CFG can disturb watermark construction. Therefore, we carry out two processes; choosing a set of candidate bytes that contribute to forming a watermark, and randomly electing watermark bytes from the candidates.

Watermarkable Zone. We determine a code region that embraces a set of candidate bytes for a watermark, that is, a *watermarkable zone* by excluding bytes unsuitable for watermarking. First, we rule out a byte if it falls into a dispatcher function[4] because it is commonly used for all smart contracts. Second, we exclude all operands since they can be more fragile to a watermark corruption (*e.g.*, modifying jumping destinations, immediate values on the stack).

Opcode Group. From a collection of all bytes (*i.e.*, list of opcodes) in the above watermarkable zone, we group a sequence of opcodes together by taking gas consumption into consideration. An *opcode group* can be determined with three factors; ① a cost threshold (T) that represents the aggregate of gas costs for consecutive opcodes, ② the size of a sliding window (W), and ③ the maximum size of a group (G) or the number of opcodes. To exemplify, in Fig. 2, the first four consecutive bytes (*e.g.*, `0x5B, 0x82, 0x01, 0x91`) at the first block table (❶) can form a group because the gas sum (1+3+3+3=10) exceeds $T = 9$ in case of $G = 5$ and $W = 1$. In the same vein, the next five consecutive bytes (*e.g.*, `0x5B, 0x82, 0x01, 0x91, 0x90`) can hold another group as it meets the requirements of the gas sum of 13 when the group size is 5. Next, we move forward (*e.g.*, starting from `0x82`) to seek the next group candidates until all blocks are covered. The reasoning behind this process is that we desire to generate as many opcode group candidates (to choose from) as possible

[4]The dispatcher is a built-in function that points to user-defined (public) functions, which should be invoked at the beginning of runtime bytecode (*e.g.*, the start function in Fig. 2).



Watermark Distribution	
Block	Byte
❶	0x52
❷	0x15
❸	0x06
❹	0x67
❺	0x10

❶ Block hash: 0xbcb02238		
Opcode (Mnemonic)		Gas
0x5B	JUMPDEST	1
0x82	DUP3	3
0x01	ADD	3
0x91	SWAP2	3
0x90	SWAP1	3
0x60	PUSH1	3
0x52	MSTORE	3
0x60	PUSH1	3
0x60	PUSH1	3
0x20	SHA3	10
0x90	SWAP1	3

❷ Block hash: 0x8d952f4d		
Opcode (Mnemonic)		Gas
0x5B	JUMPDEST	1
0x81	DUP2	3
0x54	SLOAD	10
0x81	DUP2	3
0x52	MSTORE	3
0x90	SWAP1	3
0x60	PUSH1	3
0x01	ADD	3
0x90	SWAP1	3
0x60	PUSH1	3
0x01	ADD	3
0x80	DUP1	3
0x83	DUP4	3
0x11	GT	3
0x61	PUSH2	3
0x57	JUMPI	10

❸ Block hash: 0x59e33dc6		
Opcode (Mnemonic)		Gas
0x5B	JUMPDEST	1
0x60	PUSH1	3
0x60	PUSH1	3
0x52	MSTORE	3
0x80	DUP1	3
0x60	PUSH1	3
0x52	MSTORE	3
0x60	PUSH1	3
0x60	PUSH1	3
0x20	SHA3	10
0x60	PUSH1	3
0x91	SWAP2	3
0x50	POP	2
0x90	SWAP1	3
0x50	POP	2
0x54	SLOAD	10
0x81	DUP2	3
0x56	JUMP	8

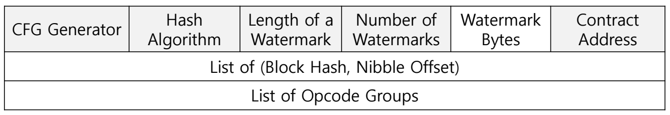
Block Hash	❶ 0xbcb02238	❷ 0x8d952f4d	❸ 0x59e33dc6	❹ 0x050ff7c4	❺ 0x9f5f873b
Byte Stream per Block	0x5B820191605260	0x5B8154906001808311	0x605280605260609150990	0x...67...	0x...10...
Nibble Offset in a Block	10	3	11	45	43

Fig. 2. Example of strategic byte election with the ERC20 runtime bytecode for a five-byte watermark. Once CFG construction is completed, we randomly elect a series of opcode groups (*i.e.*, vertical lines in the block tables) from a watermarkble zone (Section IV-B), forming a final watermarkable byte stream. This example illustrates five blocks (*i.e.*, dotted blocks; ❶-❺) where three of them demonstrate how each byte of the watermark values (on the top right table) has been distributed (*e.g.*, red nibbles) across separate blocks. The table on the bottom shows a block hash and a nibble offset in each byte stream for a block (value corresponding to the watermark), being recorded in a WRO.

while avoiding a fully overlapped group under the distribution of opcode costs (3 as a mean value)[5], the number of blocks per smart contract (108.8 on average), and the number of opcode per block (21.7 on average).

Watermark Byte Election. Concisely, once every opcode group in a watermarkable zone is set up, we randomly subset all opcode groups, followed by electing bytes for a watermark. First, we randomly choose $R\%$ of all opcode groups. The elected groups may be partially overlapped or consecutive, forming a *byte stream per block*. Notably, the byte stream serves a basis for further watermark verification. Second, we elect L bytes at random so that each byte can be present in a different block for a better spread where L is the number of distinguishing bytes that a watermark demands. As a concrete example, Fig. 2 illustrates the whole process of byte election with the ERC20 smart contract [36]. In this example, we show three watermark bytes (*e.g.*, 0x52, 0x15, 0x06) that are elected across eight opcode groups from three blocks (*e.g.*, 2, 3 and 3 per block) in case of L = 5. We adopt a unit of nibble (4 bits) to increase the likelihood of having the number of unique

[5]The most frequently appeared opcode is PUSH1 in our dataset, whose cost is three. Note that more than three out of four opcodes (76.2%; 98,137,334 of 128,819,501) consume a gas cost of three.



CFG Generator	Hash Algorithm	Length of a Watermark	Number of Watermarks	Watermark Bytes	Contract Address
List of (Block Hash, Nibble Offset)					
List of Opcode Groups					

Fig. 3. Structure of a WRO. The grey fields represent fixed-length values while others are not.

bytes from the byte stream (one may want to use more fine-grained unit like a bit). As an example, the second watermark byte of 0x15 in Fig. 2 has been taken from the second nibble at 0x81 and the first nibble at 0x54 (red letters). Finally, a nibble offset is required within a block for every elected watermark byte. Then we bookkeep the information of a watermark including opcode groups, block hashes, and nibble offsets in a pre-defined structure (Figure 3).

3) Watermark Reference Object (WRO): We define a structure, dubbed WRO, which records crucial information to validate a smart contract watermark, including block identifiers, nibble offsets within each block, opcode groups, the length of a watermark as well as the watermark itself. Note that a block identifier is a hash value of a byte stream (*i.e.*, subset of all opcodes), not of an entire block. Similarly, an offset represents a distance within a byte stream for better resiliency. Fig. 3 shows the structure of the object in detail. Additionally, it contains a CFG generation tool identifier (our technique relies on a deterministic CFG construction), a block hash algorithm (each block takes the first four bytes of the byte stream's hash for identification), the number of a watermark (multiple watermarks are embedded for robustness against partial corruptions), and a contract address. As presented in Section IV-B, we compute the hash of the WRO (*i.e.*, WRO MAC), and store it to a variable in a constructor (Listing 1).

```
contract Watermark {
    bytes WRO_MAC;
    constructor() {
        // WRO MAC with Keccak-256
        WRO_MAC = "cc860417...fc6b";
    }
}
```

Listing 1. Example of inserting a WRO MAC in Solidity. A constructor holds a variable for the MAC that resides in a creation bytecode after deployment.

C. Watermark Verification

A watermark verification entails three main phases as follow. First, given a WRO, we can compute the hash of the WRO and compare it with the stored WRO MAC extracted from the creation bytecode to ensure the integrity of the WRO. Second, given a target runtime bytecode, watermark verification reconstructs a CFG with the same CFG generator during an embedding process, followed by creating a byte stream (per block) with a list of opcode groups in the WRO. The way to seek a certain byte stream in a CFG is through a series of byte-level searches at every target block. Third, a verifier can compute every block hash with that byte stream, and find watermark bytes from each block with nibble offset information. The verifier can successfully recover a watermark unless an adversary corrupts one of the byte streams contributing to the watermark. Note that multiple watermarks are inserted to enhance SMARTMARK's resilience against corruptions.

V. IMPLEMENTATION

Our SMARTMARK prototype is written in Python 3.9. We leverage EtherSolve [37] into disassembling a smart contract bytecode and generating a CFG. If the CFG holds multiple blocks having the same bytecode, we consider only one block so that all blocks are uniquely different for SMARTMARK.

Hash Algorithm. The case that SMARTMARK employs a hash algorithm is twofold (Section IV-B): one for generating the hashes of all blocks in a CFG, and one for creating a WRO MAC. In both cases, we utilize the Keccak-256 hash algorithm whose output is a fixed length of 256 bits. Note that we take the first four bytes of a Keccak-256 digest to represent a block hash, which is equivalent to generating an identifier for a user-defined function when compiling a smart contract into bytecode in EVM.

Hyperparameters. We deliberately leave a handful of hyperparameters to be able to be adjusted (Section IV-B) in need for SMARTMARK. For determining opcode groups, we introduce a gas threshold (T) with the size of a sliding window (W) and that of an opcode group (G), which assists in electing bytes to meet the requirements (Section III-B). The ratio of elected opcode groups is set to $R\%$, which forms a byte stream per block. The length of a watermark is set to L bytes, and the current SMARTMARK implementation elects L blocks accordingly (*i.e.*, electing a single byte per block). Lastly, the number of embedded watermarks (N) is for robust verification where multiple watermarks may hold different values. It is possible to insert a different length for each watermark, however, we use the same lengths for a straightforward security analysis. The following enumerates concrete hyperparameters in our experiment for SMARTMARK: $T = 9$, $W = 1$, $G = 5$, $R = 0.2$, $N = \{1, 3, 5, 7\}$, and $L = \{10, 20\}$. In general, care must be taken in setting parameters as follows: ① N and L rely on the number of blocks in a smart contract, ② R strikes a balance between the probability of successful attacks (*e.g.*, too high R may increase a distortion attack with a higher chance of choosing precise opcode groups) and the number of opcode group candidates available (*e.g.*, too low R may fail to have sufficient candidates), and ③ we empirically advise $T = 9$ and $G = 5$ for both effectiveness and efficiency.

VI. EVALUATION

In this section, we present how well SMARTMARK meets the requirements of a smart contract watermark (Section III-B), and analyze its robustness from a security perspective (Section III-C). We evaluate SMARTMARK on a 64-bit Ubuntu 18.04 system equipped with Intel(R) Xeon(R) Gold 6230 2.10 GHz and 567GB RAM.

Dataset. We collected all fifteen million blocks (mined during the period 30 July 2015 to 21 June 2022) from the Ethereum Mainnet, which incorporates $4,112,336$ smart contracts. Then, we obtain $445,930$ unique runtime bytecodes (10.8% of the whole) after eliminating all byte-level equivalent ones. We leverage EtherSolve [37] to obtain CFGs for $284,344$ contracts, followed by taking $178,119$ contracts (62.6%) that contain a unique watermarkable zone (recall that operands and dispatcher blocks are excluded for our scheme). We indeed observe a high ratio of code duplication in smart contracts, aligned with the previous findings [6]. We manually confirm that a considerable number of code reuse cases arise from the standards [38] introduced by the Ethereum community such as ERC-20 (standard interface for fungible tokens) and ERC-721 (standard interface for non-fungible tokens). In this respect, we group analogous contracts together to demonstrate the effectiveness of our watermarking scheme. We perform DBSCAN clustering [39] with the similarity metric of $\max(\frac{len(a \cap b)}{len(a)}, \frac{len(a \cap b)}{len(b)})$ where a and b denote the CFG blocks for the two contracts, A and B. This is because SMARTMARK mainly targets disparate bytecodes (*i.e.*, program logic), excluding a common component like ERC20 that causes an overlapping between contracts. We finally obtain $27,824$ smart contracts out of 178K (15.6%) distinguishing sample contracts for our experiment. The average size of the smart contracts in our dataset is $5.8KB$ with the standard deviation of $4.6KB$ (median: 4.5 KB).

A. Imperceptibility

In SmartMark, a watermark is completely imperceptible as long as its WRO is privately kept by the smart contract owner. To embed a watermark into a smart contract, SmartMark does not add any additional code and data except a WRO MAC on its creation bytecode; the watermark is constructed with watermark bytes randomly elected from its runtime bytecode, resulting in that watermark bytes are indistinguishable from the other bytes in the runtime bytecode. Perhaps, a sophisticated adversary can identify the WRO MAC from the creation bytecode. However, we note that a WRO MAC is a cryptographic hash value that is irreversible. Therefore, one cannot obtain fruitful information about WRO from the WRO MAC.

B. Spread

Going back to Fig. 2, SmartMark intentionally picks a single byte from a byte stream per block, resulting in a well-scattered watermark across different blocks in a CFG. In this example, five out of 13 blocks (around 38%) are covered for a watermark, and a lengthier one would be more dispersed. Besides, by design, SmartMark allows one to insert multiple watermarks so that it could increase verifiability even with a single watermark being survived. Such a design choice makes SmartMark robust against varying attacks by lowering the possibility of damaging every watermark simultaneously. Oftentimes, it would be excessively costly for an adversary to disrupt a well-distributed watermark, hampering further transactions (Section VI-G).

C. Capacity

In practice, the total length (*i.e.*, capacity) of a watermark (or multiple watermarks) is bounded by the number of blocks in a CFG because our scheme relies on how to choose opcode groups in a watermarkable zone. In our experimental setting, we elect one byte from a single block. In general, the capacity of our watermarking scheme is determined by the number of watermarkable blocks available in a smart contract, rather than the size of the contract.

D. Efficiency

To demonstrate the efficiency of SmartMark, we run experiments of embedding and verification processes 10 times to measure CPU time. Note that we exclude any smart contract that does not conform to the code size limit (*i.e.*, EIP-170 [40]) in our experiments. Furthermore, we measure sub-phases for watermarking operations (Table I); the phase of electing watermark bytes dominates the entire embedding time (96.7%) whereas that of creating (watermarkable) bytestream does the verification time (70.1%). A wide range of variations mainly arise from processing time that largely depends on the number of blocks pertaining to a watermark, rather than the size of a smart contract. We observe that a verification process (mean: $17,258$ milliseconds) takes approximately 1.5 times longer than an embedding process (mean: $11,088$ milliseconds), but is still overall acceptable (*e.g.*, within $20,000$ milliseconds) in practice.

TABLE I
BREAKDOWN OF EMBEDDING AND VERIFICATION TIME ON AVERAGE. ELECTING BYTES FOR A WATERMARK AND CREATING A BYTESTREAM DOMINATE EMBEDDING AND VERIFICATION TIME, RESPECTIVELY.

Process	Phase	Ratio (%)	Time (ms)
Embedding	Opcode grouping	3.38	363.88 ± 410.62
	Watermark byte election	96.69	$10,720.59 \pm 13,083.91$
	WRO creation	0.03	3.38 ± 2.49
Verification	WRO verification	0.01	0.71 ± 0.70
	Bytestream creation	70.06	$12,089.79 \pm 14,421.60$
	Hash discovery	29.93	$5,165.84 \pm 5,097.88$

TABLE II
RATIO OF UNIQUE WATERMARKS BY THE WHOLE SIZE ($L \times N$) OF AN EMBEDDED WATERMARK. L AND N DENOTE THE LENGTH OF A WATERMARK (L) AND THE NUMBER (N) OF WATERMARKS, RESPECTIVELY. A WATERMARKABLE CONTRACT REPRESENTS THE CONTRACT THAT INCORPORATES A SUFFICIENT NUMBER OF WATERMARKABLE ZONES.

L	N	Watermark Size (bytes)	# Unique Watermarks (Watermarkable Contracts)	Ratio
10	1	10	27,704 (27,823)	99.995%
	3	30	23,062 (23,378)	99.986%
	5	50	18,098 (18,511)	99.978%
	7	70	15,140 (15,620)	99.970%
15	1	15	27,764 (27,764)	100.000%
	3	45	19,482 (19,485)	99.999%
	5	75	15,144 (15,156)	99.999%
	7	105	13,009 (13,030)	99.998%
20	1	20	26,679 (26,679)	100.000%
	3	60	16,874 (16,874)	100.000%
	5	100	13,311 (13,311)	100.000%
	7	140	11,646 (11,646)	100.000%

E. Cost

Recall that SmartMark does not introduce any additional routine on a runtime bytecode, staying the original execution gas intact. Instead, SmartMark increases a (one-time) creation cost due to a WRO MAC in a constructor (Listing 1 in Section IV-B) where its cost is closely proportional to the size of the WRO MAC. In case of embedding a 256-bit WRO MAC using Keccak-256 in our implementation, an additional cost ranges from 48,540 to 53,100 gas. Such cost variation mostly arises from varying opcodes depending on the original context of the constructor when inserting a WRO MAC. Although a gas price frequently fluctuates, as of writing, the additional gas consumption is around $4.27 \sim 4.67$ US dollars (USD) with the exchange rate of $90 \sim 100$ gas per bit.

F. Credibility

Credibility is one of the essential requirements for a watermark scheme, which ensures that it can be reliably extracted for the proof of an ownership. We assess SmartMark by confirming that a watermark from WRO must be unique, that is, the watermark cannot be present elsewhere but the original contract. Table II summarizes the total bytes of watermark(s) ($L \times N$) with the length (L) and the number (N) of the watermark(s) inserted into a smart contract, and the ratio of contracts that hold unique watermarks accordingly. Due to the constraint of an embedding watermark size that relies on a smart contract size, we compute the ratio of uniqueness based on the number of the watermarkable contracts. For example, a single 10-byte long watermark can be possibly inserted into $27,823$ (99.999%) out of $27,824$ contracts in total, resulting that $27,704$ holds unique values (99.995%). Empirically, the uniqueness ratio is proportional to the length of a watermark whereas inversely proportional to the number of the watermark. Hence, we advise to strike a balance between the length and the number for fulfilling both capacity and spread properties (as well as credibility).

G. Resiliency

In this section, we show how SMARTMARK can defend against the attacks presented in Section III-C. Note that Section VI-A covers *unauthorized recognition*.

1) Collusion: A collusive attack would help identify a smart contract's WRO MAC by analyzing the differences between the creation bytecodes of several smart contracts because all WRO MAC values always have the same fixed length, *e.g.*, SHA-256 produces a 32 bytes hash value, and have a higher entropy than the other variable values. As mentioned in Section VI-A, however, we note that the presence of a WRO MAC itself does not practically help recognize its corresponding watermark because a WRO MAC is a cryptographic hash value that is irreversible.

2) Addition: We note that an adversary can embed a watermark W_a of one's choice into a smart contract using SMARTMARK in the same manner even when a watermark W_o is already present in a smart contract. In such a case, one can verify the presence of both watermarks (W_a and W_o) with the original smart contract owner's own WRO and the adversary's own WRO, respectively. However, it is easily recognizable that W_o was embedded prior to W_a on the Ethereum network.

3) Deletion: With our scheme, a watermark is constructed with existing opcodes in an original smart contract. In theory, a watermark deletion may be possible when an adversary could replace one or more opcodes used for the watermark with others, however, such semantic-preserving code transformation that maintains a reasonable cost would be quite challenging.

4) Distortion: One of plausible and powerful attacks to corrupt our watermark scheme is a distortion attack with arbitrary transformations. We randomly choose 2,500 smart contracts whose source code have been publicly available. In this experiment, we set the length of a watermark (L) to be 15, and the number of watermarks (N) to be 3.

Empirical Results. We conduct various distortion experiments for thwarting an embedded watermark with the following five different types of transformations as suggested by Chen et al. [6]: ① adjusting a function visibility (*e.g.*, access modifier alters `public` to `private`, or the other way around), ② updating an inheritance relationship (*e.g.*, arbitrary subcontract is added and inherited), ③ introducing an additional state variable (*e.g.*, original state variable refers to an arbitrary state variable), ④ defining a new event and function (*e.g.*, additional function has been added to an original contract), and ⑤ adding a statement (*e.g.*, arbitrary statement is added to an original function). Table III summarizes the results of the above distortion attempts against our scheme. SMARTMARK shows the robustness of an individual attack (*i.e.*, 99% or above), and applying all transformations barely drops the ratio of appropriate verification (*i.e.*, 98.9%).

TABLE III
EXPERIMENTAL RESULTS OF VARYING DISTORTION ATTACKS AGAINST AN EMBEDDED WATERMARK ON 2,500 SMART CONTRACTS, WHICH SHOWS THE ROBUSTNESS OF THE SMARTMARK SCHEME.

Transformation Type (Distortion)	# Verified Contracts	Ratio
① Adjusting a function visibility	2,490	99.60%
② Updating an inheritance	2,495	99.80%
③ Introducing an additional state variable	2,489	99.56%
④ Defining an event & function	2,500	100.00%
⑤ Adding a statement	2,475	99.00%
⊕ Applying all the above	2,472	98.88%

Theoretical Analysis. As presented in Section VI-A, an adversary cannot distinguish watermark bytes from other bytes in a smart contract. Therefore, for each block in the CFG of a target contract containing a watermark byte, the best distortion strategy at the bytecode level would be guessing an opcode group used for the watermark and adding it to the block. If the adversary's guess is correct, the newly added opcode group would be used to form a watermarkable byte stream with the existing opcode groups for the watermark to compute the CFG block hash. Consequently, the valid block hash contained in the WRO would not match the new block hash, leading a watermark corruption.

Given a smart contract of s, the probability of an attack success ($P_{attack}\ (L, B_s, M_s)$) with a distortion can be computed by Equation (1) where L, B_s, and M_s represent the length of the watermark, the number of candidate blocks (in a watermarkable zone) for s, and the number of candidate blocks (in a watermarkable zone) that has been modified by an adversary against s, respectively. Figure 4 concisely illustrates those parameters. Watermark bytes are scattered in candidate blocks in a watermarkable zone. Suppose that an adversary was able to modify the block(s) of one's choice where some of which could contain a watermark byte.

$$P_{attack}(L, B_s, M_s) = \frac{\sum_{i=1}^{min(L,M_s)} \binom{B_s}{M_s}\binom{M_s}{i}\binom{B_s-M_s}{L-i}}{\binom{B_s}{L}\binom{B_s}{M_s}} \quad (1)$$

In Equation (1), the denominator represents the number of all possible ways by choosing ① L blocks from B_s blocks disallowing duplicates, and ② M_s blocks from B_s blocks disallowing duplicates, respectively. The numerator represents the number of successful distortion attacks. When an adversary modifies M_s blocks from B_s blocks, the distortion attack would be successfully performed if those modified blocks contain at least one block that contains watermark bytes. In Equation (1), i represents the number of blocks that contains actual watermark bytes out of M_s blocks modified by an adversary. In this scenario, the number of successful distortion attacks can be interpreted as the number of possible ways to choose M_s blocks from B_s blocks, i blocks from those M_s

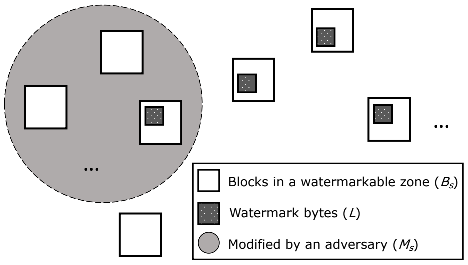

Blocks in a watermarkable zone (B_s)
Watermark bytes (L)
Modified by an adversary (M_s)
...

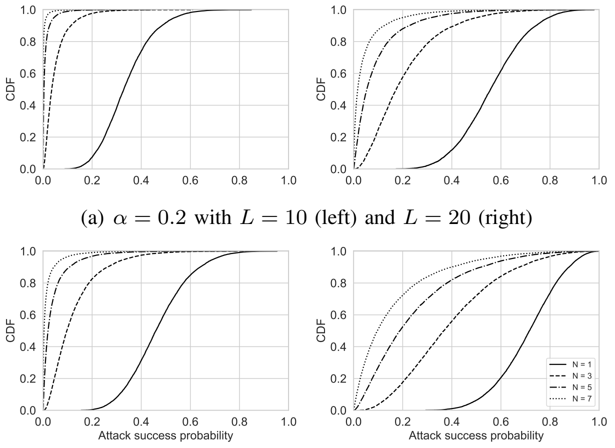
Fig. 4. Illustrative parameters to compute the probability of an attack success, $P_{attack}(L, B_s, M_s)$. An empty rectangle represents a candidate block in a watermarkable zone, where the rectangle with a dark box contains a watermark byte. The dotted circle represents an area that includes the candidate block(s) modified by an adversary.

blocks, and $L - i$ blocks from the remaining $B_s - M_s$ blocks disallowing duplicates. Finally, $\binom{B_s}{M_s}$ can be canceled out in the numerator and the denominator.

To enhance the resiliency of SMARTMARK against distortion attacks, we can insert N watermarks that do not overlap each other where $N > 1$. In this case, an adversary needs to corrupt all N watermarks for a successful attack. We presume that all watermarks have the same length for simplicity of analysis. With the notation of $P_{attack}(L, B_s, M_s, N)$, the probability of an attack success with N watermarks on a smart contract s. $P_{attack}(L, B_s, M_s, N)$ can be simply expanded from $P_{attack}(L, B_s, M_s)$ as in Equation (2).

$$P_{attack}(L, B_s, M_s, N) = P_{attack}(L, B_s, M_s)^N \qquad (2)$$

We compute $P_{attack}(L, B_s, M_s, N)$ for each smart contract in the 27,824 smart contracts with varying parameters such as L, N, and α where α represents the ratio of the maximum allowable gas cost for the opcodes added by an adversary (hereafter referred to as "attack budget") to the total gas cost to execute a target smart contract. To compute $P_{attack}(L, B_s, M_s, N)$, we need to concretely obtain B_s and M_s from the smart contract s. With the hyperparameters of $T = 9$, $W = 1$, $G = 5$, and $R = 0.2$, as watermarkable blocks for s are determined by our SMARTMARK implementation, we can empirically obtain B_s. However, M_s cannot be determined by SMARTMARK because M_s depends on an adversary's choice – an adversary needs to correctly guess an opcode group for a candidate block containing watermark bytes and add it to the block. Therefore, given a smart contract s, we compute the expected value of M_s ($E(M_s)$) with specific parameter values for s. The attack budget can be computed as $\alpha \cdot \Psi_s$ where Ψ_s represents the total gas cost to execute the smart contract s. For example, when $\Psi_s = 1,000$ and $\alpha = 0.5$, the attack budget would be 500. From the attack budget $\alpha \cdot \Psi_s$, we can compute the maximum number of opcode groups added to s for a distortion attack as $\lfloor \alpha \cdot \Psi_s / T \rfloor$ because the gas cost for each opcode group, which corrupts a watermark byte effectively, is greater than or equal to T. In the previous example, when $T = 9$, the maximum number of opcode groups added to s is 55. We assume that an adversary knows



(a) $\alpha = 0.2$ with $L = 10$ (left) and $L = 20$ (right)

(b) $\alpha = 0.3$ with $L = 10$ (left) and $L = 20$ (right)

Fig. 5. Cumulative distribution functions (CDFs) of $P_{attack}(L, B, M, N)$ with L = {10, 20} and N = {1, 3, 5, 7} for 2,738 smart contracts.

T used for SMARTMARK for simplicity of analysis. Then, the adversary's best attack strategy is to select $\lfloor \alpha \cdot \Psi_s / T \rfloor$ blocks out of C_s blocks and add an opcode group to each block one by one where C_s represents the number of the blocks that contains opcode groups, which requires a gas cost greater than or equal to T. With this strategy, the probability of selecting a candidate block in a watermarkable zone is B_s/C_s whenever an adversary modifies a block. Thus, $E(M_s)$ can be computed by multiplying $\lfloor \alpha \cdot \Psi_s / T \rfloor$ by B_s/C_s as follow:

$$E(M_s) = \left\lfloor \frac{\alpha \cdot \Psi_s}{T} \right\rfloor \cdot \frac{B_s}{C_s} \qquad (3)$$

Figure 5 shows the cumulative distribution functions (CDFs) when computing $P_{attack}(L, B, M, N)$ with α = {0.2, 0.3}, L = {10, 20}, and N = {1, 3, 5, 7} with the selected 11,646 smart contracts for fair comparison (recall that the number of watermarkable contracts may be different depending on L and N). Overall, choosing either a smaller L or a larger N raises the bar by requiring a higher execution cost for successful attacks, confirming our intuition on the resiliency of SMARTMARK against distortion attacks. With $L = 10$, $N = 7$, and $\alpha = 0.2$, only 0.7 of the smart contracts would be thwarted with the attack success probability of 0.05 or higher. Even when α increases 0.3 with the same configuration, only 8.9% would be thwarted, being still effective. On the contrary, with $L = 20$, $N = 7$, and $\alpha = 0.2$, the attack success probability significantly increases; 25.3% would be thwarted with the attack success probability of 0.05 or higher. Based on these results, we advise not to utilize $L = 20$ despite its superiority in credibility (Section VI-F). Likewise, a single watermark (when N = 1) would be ineffective against distortion attacks, indicating that multiple watermarks are recommended for SMARTMARK. It is noted that an adversary cannot increase α unreasonably because generated smart contracts with a large α is not competitive at all against the original smart contract in terms of the execution cost.

VII. Discussions and Limitations

This section covers in-depth discussions and limitations of our approach, and future research.

Reversing EVM Bytecode. EVM is a Turing complete virtual machine based on the stack, which does not follow a von Neumann architecture (*e.g.*, no registers). Besides, Solidity even complicates bytecodes by introducing built-in functions and applying various optimizations. In this regard, reverse engineering of an EVM bytecode and understanding underlying semantics (*i.e.*, decompilation) are non-trivial [41], [42], [43]. This would make distortion attacks without source code quite challenging on a smart contract.

EVM Bytecode Diversification. Having the identical source code, a runtime EVM bytecode may be diversified with a rapid evolution of the Solidity compiler [44]. This is mostly because of selecting cost-effective instructions. The current version of SmartMark is limited solely when generating bytecodes with the same compiler version. Supporting various compiler versions is part of our future work.

Threats to Validity. The threats to the validity of this work mainly come from two aspects. A possible threat is whether we used representative smart contracts for evaluation. Even though we collected 4,112,336 smart contracts from all Ethereum blocks, our experiments includes only 27,824 smart contracts based on the DBSCAN clustering results for distinct smart contracts. We have excluded small-sized smart contracts from our dataset that may not have a sufficient number of blocks for watermark insertion. However, it would not be problematic with considerably complex business logics in most cases, which increases the size (and the number of blocks accordingly) of a smart contract. Another threat to validity is the generalizability of SmartMark. Because the current implementation of SmartMark relies on EtherSolve [37] to generate CFGs from smart contracts, our evaluation may not be applicable with a different CFG generator. Supporting additional CFG generators is part of our future work.

VIII. Related Work

Software Copyright and Smart Contracts. Protecting a software copyright often helps to maintain productivity and motivation of software development. Vast studies [45], [46], [47] have been conducted on the methods to detect and/or prevent a software piracy. Lately, the rise in popularity of smart contracts with the blockchain technology necessitates a new suitable means to safeguarding an ownership [48], [49].

Smart Contract Code Reuse. Recent studies repeatedly show that reusing code in a smart contract is quite prevalent [50], [51], [52], [6], [5]. Chen et al. [6] reveal that 91.1% out of 52,951 smart contract projects gathered by Etherscan [53] (before August 2019) contain one or more subcontracts from others. According to the analysis by Pierro et al. [5], a wide adoption of code reuse in a smart contract arises from the developers' desire for building successful Ethereum DApps and/or the lack of a well-integrated development tool. While code reusability aids the quick and effortless development of a smart contract, a security bug may bring about an unwelcome outcome [1] as seen in the past incidents [54], [55], [56], [57]. In the meantime, EClone [44] detects a replication of a smart contract based on its birthmark. Note that SmartMark is designed for embedding and verifying a watermark while EClone aims to measure the similarity between smart contracts without considering security.

Software Watermarking Schemes. A wide spectrum of software watermarking techniques [8], [9], [10], [11], [7], [12], [13], [15], [14], [16], [17], [18], [19], [20], [21], [22] have been proposed to protect a software copyright. One of simple but efficient approaches leverages code reordering [10], [11] at the level of a basic block [10] or a function [11], which inserts a watermark by mapping it into the order of code. Another well-studied direction for software watermarking utilizes a graph theory [7], [12], [13], [14] such as a graph coloring problem [13]. Collberg et al. [7] store a graph structure on the heap at runtime. Meanwhile, an obfuscation scheme has been widely adopted [15], [16], [17], [18] in the field of software watermarking, which includes inserting dummy methods and opaque predicates [16], and steganography [17]. Besides, the idea of embedding a watermark into a target application without modification (*i.e.*, zero watermarking [19], [20], [21]) has been introduced, however, its downside lies in needing additional storage for bookkeeping. Meanwhile, Ma et al. [22] introduce an return-oriented-programming (ROP) based watermarking scheme, which inserts a well-crafted code to be triggered into a data region for verification afterward. However, applying prior software watermarking schemes to a smart contract is impractical due to its unique properties such as the restriction of code size (*e.g.*, code relocation, obfuscation), the absence of dynamic allocation (*e.g.*, runtime operation), and execution costs (*e.g.*, dummy code insertion). SmartMark proposes a distinct watermarking scheme tailored to smart contracts for the first time by addressing the above hindrances.

IX. Conclusion

Smart contracts fundamentally have different characteristics from conventional programs. The difference induces several restrictions on adopting existing software watermark techniques to smart contracts. In this work, we present SmartMark, a novel watermarking scheme on smart contracts. Our empirical evaluation shows the practicality and effectiveness of SmartMark from both security and economic perspectives, which is resistant to various attacks against a watermark at acceptable cost.

X. Data and Source Code Availability

To foster further watermarking research for smart contracts, we disclose all of our evaluation dataset to the public. We have opened the datasets on a preserved digital repository[6] and source code[7], so that anyone can reproduce our work.

[6]https://figshare.com/s/884a0de3ab867803549c

[7]https://github.com/smartmarker/SmartMark

REFERENCES

[1] N. He, L. Wu, H. Wang, Y. Guo, and X. Jiang, "Characterizing Code Clones in the Ethereum Smart Contract Ecosystem," in *Proceedings of the 24th International Conference on Financial Cryptography and Data Security (FC)*, 2020, pp. 654–675.

[2] Y. Zhou, D. Kumar, S. Bakshi, J. Mason, A. Miller, and M. Bailey, "Erays: Reverse Engineering Ethereum's Opaque Smart Contracts," in *Proceedings of the 27th USENIX Security Symposium (Security)*, 2018, pp. 1371–1385.

[3] L. Brent, A. Jurisevic, M. Kong, E. Liu, F. Gauthier, V. Gramoli, R. Holz, and B. Scholz, "Vandal: A Scalable Security Analysis Framework for Smart Contracts," *arXiv preprint arXiv:1809.03981*, 2018.

[4] N. Grech, L. Brent, B. Scholz, and Y. Smaragdakis, "Gigahorse: Thorough, Declarative Decompilation of Smart Contracts," in *Proceedings of the 41st IEEE/ACM International Conference on Software Engineering (ICSE)*, 2019, pp. 1176–1186.

[5] G. A. Pierro and R. Tonelli, "Analysis of Source Code Duplication in Ethereum Smart Contracts," in *Proceedings of the 28th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2021, pp. 701–707.

[6] X. Chen, P. Liao, Y. Zhang, Y. Huang, and Z. Zheng, "Understanding Code Reuse in Smart Contracts," in *Proceedings of the 28th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2021, pp. 470–479.

[7] C. Collberg and C. Thomborson, "Software Watermarking: Models and Dynamic Embeddings," in *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 1999, pp. 311–324.

[8] P. R. Samson, "Apparatus and Method for Serializing and Validating Copies of Computer Software," http://www.google.com/patents/US5287408A, 1994.

[9] K. Holmes, "Computer Software Protection," http://www.google.com/patents/US5287407, 1994.

[10] R. I. Davidson and N. Myhrvold, "Method and System for Generating and Auditing a Signature for a Computer Program," https://patents.google.com/patent/US5559884A, 1996.

[11] H. Kang, Y. Kwon, S. Lee, and H. Koo, "SoftMark: Software Watermarking via a Binary Function Relocation," in *Proceedings of the 37th Annual Computer Security Applications Conference (ACSAC)*, 2021, pp. 169–181.

[12] Z. Jiang, R. Zhong, and B. Zheng, "A Software Watermarking Method Based on Public-key Cryptography and Graph Coloring," in *Proceedings of the 3rd International Conference on Genetic and Evolutionary Computing (ICGEC)*, 2009, pp. 433–437.

[13] G. Qu and M. Potkonjak, "Analysis of Watermarking Techniques for Graph Coloring Problem," in *Proceedings of the 1998 IEEE/ACM international Conference on Computer-Aided Design (ICCAD)*, 1998, pp. 190–193.

[14] ——, "Hiding Signatures in Graph Coloring Solutions," in *Proceedings of the 3rd Workshop on Information Hiding (IH)*, 1999, pp. 348–367.

[15] V. Balachandran, N. W. Keong, and S. Emmanuel, "Function Level Control Flow Obfuscation for Software Security," in *Proceedings of the 8th International Conference on Complex, Intelligent and Software Intensive Systems (CISIS)*, 2014, pp. 133–140.

[16] A. Monden, H. Iida, K. Matsumoto, K. Inoue, and K. Torii, "A Practical Method for Watermarking Java Programs," in *Proceedings of the 24th Annual International Computer Software and Applications Conference (COMPSAC)*, 2000, pp. 191–197.

[17] K. Lu, S. Xiong, and D. Gao, "Ropsteg: Program Steganography with Return Oriented Programming," in *Proceedings of the 4th ACM Conference on Data and Application Security and Privacy (CODASPY)*, 2014, pp. 265–272.

[18] C. Collberg, G. Myles, and A. Huntwork, "Sandmark-A Tool for Software Protection Research," *IEEE Security & Privacy (S&P)*, pp. 40–49, 2003.

[19] J. Zuo and D. Cui, "Zero-watermark resistant to mp3 compression," *Advanced Materials Research*, pp. 254–259, 2010.

[20] A. Roček, M. Javorník, K. Slavíček, and O. Dostál, "Zero Watermarking: Critical Analysis of its Role in Current Medical Imaging," *Journal of Digital Imaging*, pp. 204–211, 2021.

[21] C. Wang, X. Wang, Z. Xia, and C. Zhang, "Ternary Radial Harmonic Fourier Moments Based Robust Stereo Image Zero-watermarking Algorithm," *Information Sciences*, pp. 109–120, 2019.

[22] H. Ma, K. Lu, X. Ma, H. Zhang, C. Jia, and D. Gao, "Software Watermarking Using Return-oriented Programming," in *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security (CCS)*, 2015, pp. 369–380.

[23] C. Collberg, E. Carter, S. Debray, A. Huntwork, J. Kececioglu, C. Linn, and M. Stepp, "Dynamic Path-based Software Watermarking," in *Proceedings of the 25th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2004, pp. 107–118.

[24] M. Zhang, P. Zhang, X. Luo, and F. Xiao, "Source Code Obfuscation for Smart Contracts," in *Proceedings of the 27th Asia-Pacific Software Engineering Conference (APSEC)*, 2020, pp. 513–514.

[25] Solidity, "Solidity — Solidity 0.8.13 Documentation," https://docs.soliditylang.org/en/v0.8.13/, 2022.

[26] W. Yan, J. Gao, Z. Wu, Y. Li, Z. Guan, Q. Li, and Z. Chen, "Eshield: Protect Smart Contracts against Reverse Engineering," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2020, pp. 553–556.

[27] N. Szabo, "Formalizing and Securing Relationships on Public Networks," *First Monday*, 1997.

[28] G. Wood *et al.*, "Ethereum: A Secure Decentralised Generalised Transaction Ledger," *Ethereum Project Yellow Paper*, pp. 1–32, 2014.

[29] Solidity, "Layout of a Solidity Source File, year = 2022," https://docs.soliditylang.org/en/v0.8.11/layout-of-source-files.html.

[30] Ethereum Foundation, "Opcodes for the EVM," https://ethereum.org/en/developers/docs/evm/opcodes/, 2021.

[31] N. Atzei, M. Bartoletti, and T. Cimoli, "A Survey of Attacks on Ethereum Smart Contracts (SoK)," in *Proceedings of the 6th International Conference on Principles of Security and Trust (POST)*, 2017, p. 164–186.

[32] G. Myles, C. Collberg, Z. Heidepriem, and A. Navabi, "The Evaluation of Two Software Watermarking Algorithms," *Software: Practice and Experience*, pp. 923–938, 2005.

[33] Y. Zeng, F. Liu, X. Luo, and C. Yang, "Robust Software Watermarking Scheme Based on Obfuscated Interpretation," in *Proceedings of the 2nd International Conference on Multimedia Information Networking and Security (MINES)*, 2010, pp. 671–675.

[34] M. Dalla Preda and M. Pasqua, "Software Watermarking: A Semantics-based Approach," *Electronic Notes in Theoretical Computer Science*, pp. 71–85, 2017.

[35] A. Dey, S. Bhattacharya, and N. Chaki, "Software Watermarking: Progress and Challenges," *INAE Letters*, pp. 65–75, 2019.

[36] Ethereum Foundation, "Understanding the ERC-20 Token Smart Contract," https://ethereum.org/en/developers/tutorials/understand-the-erc-20-token-smart-contract/, 2020.

[37] F. Contro, M. Crosara, M. Ceccato, and M. Dalla Preda, "EtherSolve: Computing an Accurate Control-flow Graph from Ethereum Bytecode," in *Proceedings of the 29th IEEE/ACM International Conference on Program Comprehension (ICPC)*, 2021, pp. 127–137.

[38] Ethereum Foundation, "Ethereum-Development Standards," https://ethereum.org/en/developers/docs/standards/, 2021.

[39] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu, "A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise," in *Proceedings of the 2nd International Conference on Knowledge Discovery and Data Mining (KDD)*, 1996, pp. 226—-231.

[40] Ethereum Foundation, "EIP-170: Contract Code Size Limit," https://eips.ethereum.org/EIPS/eip-170, 2022.

[41] RET2 Systems, "Building up from the Ethereum Bytecode," https://blog.ret2.io/2018/05/16/practical-eth-decompilation/, 2022.

[42] F. Sakharov, "Reversing evm bytecode with radare2," https://blog.positive.com/reversing-evm-bytecode-with-radare2-ab77247e5e53, 2022.

[43] B. Arvanaghi, "Reversing Ethereum Smart Contracts," https://arvanaghi.com/blog/reversing-ethereum-smart-contracts/, 2022.

[44] H. Liu, Z. Yang, Y. Jiang, W. Zhao, and J. Sun, "Enabling Clone Detection for Ethereum via Smart Contract Birthmarks," in *Proceedings of the 27th IEEE/ACM International Conference on Program Comprehension (ICPC)*, 2019, pp. 105–115.

[45] A. G. Peace, D. F. Galletta, and J. Y. Thong, "Software Piracy in the Workplace: A Model and Empirical Test," *Journal of Management Information Systems*, pp. 153–177, 2003.

[46] P. Samuelson, "Functionality and Expression in Computer Programs: Refining the Tests for Software Copyright Infringement," *Berkeley Tech. LJ*, p. 1215, 2016.

[47] K. Reavis Conner and R. P. Rumelt, “Software Piracy: An Analysis of Protection Strategies,” *Management Science*, pp. 125–139, 1991.

[48] A. Savelyev, “Copyright in the Blockchain Era: Promises and Challenges,” *Computer Law & Security Review*, pp. 550–561, 2018.

[49] B. Bodó, D. Gervais, and J. P. Quintais, “Blockchain and Smart Contracts: The Missing Link in Copyright Licensing?” *International Journal of Law and Information Technology*, pp. 311–336, 2018.

[50] N. Jia, Q. Kong, and H. Huang, “How Similar are Smart Contracts on the Ethereum?” in *Proceedings of the 2nd International Conference on Blockchain and Trustworthy Systems (BlockSys)*, 2020, pp. 403–414.

[51] M. Kondo, G. A. Oliva, Z. M. J. Jiang, A. E. Hassan, and O. Mizuno, “Code Cloning in Smart Contracts: a Case Study on Verified Contracts from the Ethereum Blockchain Platform,” *Empirical Software Engineering*, pp. 4617–4675, 2020.

[52] L. Kiffer, D. Levin, and A. Mislove, “Analyzing Ethereum’s Contract Topology,” in *Proceedings of the 18th Internet Measurement Conference (IMC)*, 2018, pp. 494–499.

[53] Etherscan, “Ethereum (ETH) Blockchain Explorer,” https://etherscan.io/, 2022.

[54] N. Popper, “A Hacking of More Than $50 Million Dashes Hopes in the World of Virtual Currency,” https://www.nytimes.com/2016/06/18/business/dealbook/hacker-may-have-removed-more-than-50-million-from-experimental-cybercurrency-project.html, 2016.

[55] R. Browne, “‘Accidental’ Bug May Have Frozen $280 Million Worth of Digital Coin Ether in a Cryptocurrency Wallet,” https://www.cnbc.com/2017/11/08/accidental-bug-may-have-frozen-280-worth-of-ether-on-parity-wallet.html, 2017.

[56] B. Dale, “Victims of $30M Parity Wallet Hack Offer Attacker $60M ‘Bounty’,” https://finance.yahoo.com/news/victims-30m-parity-wallet-hack-170550649.html, 2021.

[57] T. M, “Pigeoncoin – The Coin That Couldn’t Fly,” https://bitcoin.co.uk/pigeoncoin-the-coin-that-couldnt-fly, 2018.