SoftMark: Software Watermarking via a Binary Function Relocation

Honggoo Kang
Korea University
Seoul, South Korea
honggoonin@korea.ac.kr

Yonghwi Kwon
University of Virginia
Charlottesville, Virginia, USA
yongkwon@virginia.edu

Sangjin Lee
Korea University
Seoul, South Korea
sangjin@korea.ac.kr

Hyungjoon Koo*
Sungkyunkwan University
Suwon, South Korea
kevin.koo@skku.edu

ABSTRACT

The ease of reproducibility of digital artifacts raises a growing concern in copyright infringement; in particular, for a software product. Software watermarking is one of the promising techniques to verify the owner of licensed software by embedding a digital fingerprint. Developing an ideal software watermark scheme is challenging because i) unlike digital media watermarking, software watermarking must preserve the original code semantics after inserting software watermark, and ii) it requires well-balanced properties of credibility, resiliency, capacity, imperceptibility, and efficiency. We present SoftMark, a software watermarking system that leverages a function relocation where the order of functions implicitly encodes a hidden identifier. By design, SoftMark does not introduce additional structures (*i.e.*, codes, blocks, or subroutines), being robust in unauthorized detection, while maintaining a negligible performance overhead and reasonable capacity. With various strategies against viable attacks (*i.e.*, static binary re-instrumentation), we tackle the limitations of previous reordering-based approaches. Our empirical results demonstrate the practicality and effectiveness by successful embedding and extraction of various watermark values.

CCS CONCEPTS

• **Security and privacy → Software security engineering**.

KEYWORDS

Software Watermarking, Watermark, Function Reordering, Function Relocation, Binary Instrumentation

ACM Reference Format:
Honggoo Kang, Yonghwi Kwon, Sangjin Lee, and Hyungjoon Koo. 2021. SoftMark: Software Watermarking via a Binary Function Relocation. In *Annual Computer Security Applications Conference (ACSAC '21), December 6–10, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3485832.3488027

*Corresponding author

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ACSAC '21, December 6–10, 2021, Virtual Event, USA
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8579-4/21/12…$15.00
https://doi.org/10.1145/3485832.3488027

1 INTRODUCTION

Today, a vast usage of digital data makes our life convenient by sharing them with others due to its trivial reproducibility by nature. However, the ease of both data duplication and distribution raises unfavorable consequences, that is, copyright infringement when digital contents (*e.g.*, pictures, movies, TV episodes, software) are illegally copied, distributed, or publicly presented without the owner's permission. The number of disputes over copyrights on pirated materials gradually increases; in particular, software piracy is a significantly growing concern. According to the survey conducted by BSA [10], 37% of the whole software around the globe have been estimated as illegitimate or unlicensed with the commercial value of $46.3 billion.

Digital watermarking is one of promising techniques for recognizing the originality of digital works. It covertly inserts a unique digital fingerprint into digital contents such as text, image, audio, and video so that the ownership of the contents can be identified by revealing the embedded fingerprint. In a similar vein, software watermarking is a technique that aims to provide the digital fingerprint of a software product by inserting certain information that represents its owner or distributor. Then, the identifier of every software copy offers the traceability because it belongs to a unique customer upon the purchase of software.

Software watermarking is effective against an adversary who wants to run a copyrighted program free of charge, revealing neither the identity of the attacker nor the original owner of the program. The adversary may attempt to reverse-engineer a watermark embedding process as well as unauthorized detection. While it is nearly impossible to achieve complete prevention against all viable attacks, a desirable software watermarking scheme should be able to provide a sufficient level of stealthiness and resilience that renders such attacks extremely expensive, or severely discourages attackers. To this end, as with previous work [14, 21, 23, 27, 42, 49, 67], we identify key properties (requirements) of software watermarking techniques: *Credibility*, *Capacity*, *Imperceptibility*, *Resiliency*, *Spread*, and *Efficiency* (See §2 in detail).

For the last few decades, diverse software watermarking [23, 28, 70] approaches have been proposed, including reordering-based [22, 30, 51, 53, 55], graph-based [14, 18, 32, 48, 49, 69],

obfuscated-based [4, 5, 11, 15, 33, 41, 66], and branch-based [26] approaches. Depending on where/how the watermark is inserted and veri ed, software watermarking techniques in the literature can be classi ed into static [4, 22, 30, 32, 41, 48, 49, 51, 53, 55] or dynamic [14, 18] approaches. A static watermarking technique does not need to run a program whereas a dynamic watermarking technique extracts a watermark at runtime. Unfortunately, we observe that the existing approaches have di culty in achieving desirable balances between the properties, particularly resiliency and imperceptibility.

In this paper, we propose SoftMark , a software watermarking technique on top of a function relocation scheme. Since reliable relocation of binary functions is extremely challenging, our technique is highly resilient against varying attacks including unauthorized detection, illegal corruption and collusion. Moreover, SoftMark is implicitly encoded, leveraging the location of pre-selected functions in a target program; each order of the functions maps into a secret identi er.

Our watermarking scheme has addressed several drawbacks of previous reordering based approaches [22, 42] by adopting fruitful strategies that impose signi cant challenges to watermarking corruption techniques via static binary instrumentation. First, SoftMark does not introduce any codes, blocks, or subroutines to a target program, which empirically demonstrates negligible runtime and space overheads. Second, the presence of a watermark is di cult to reveal by a statistical analysis or inference unless multiple instances are collusively collected. Third, SoftMark conveys a relatively high capacity for watermark encoding, which is proportional to the size of a program (i.e., A set of n functions can represent up to $\lfloor \log_2 n! \rfloor$ bits). Fourth, the design of SoftMark shows a reasonable resiliency even under semantic-preserving code transformation by inserting multiple watermarks across a broad spectrum of functions. It is noteworthy mentioning that we select a set of unique functions with a variety of strategies that make reliable code transformation challenging. To implement SoftMark , we employ CCR [35], a special compiler toolchain that emits metadata for instrumenting a variant with a watermark.

In summary, we make the following contributions:

- We propose SoftMark , an e cient watermarking system via a function relocation based encoding, resolving most of the prior limitations.
- We have designed and implemented a prototype of SoftMark to meet the requirements of a practical software watermarking technique against various viable attacks.
- We experimentally evaluate SoftMark with real world applications, demonstrating the e ectiveness and practicality of our approach.

The source code of SoftMark will be publicly available in the near future to foster further watermarking research.

2 SOFTWARE WATERMARKING

In this section, we discuss the de nition, requirements, existing approaches and threat model of software watermarking.

2.1 Problem De nition

The objective of software watermarking is to provide a reliable identi cation service to be able to claim the ownership of a software product. In a nutshell, software watermarking consists of two separate processes: i) embedding a unique signature and ii) extracting the signature for veri cation. Formally, the processes of software watermarking are de ned as follows:

De nition 1. Given an original program $P(\cdot)$ and a watermark (W), software watermarking consists of two functions; i) a watermark embedder function is $F_{embed}(P, W) = P_W$ where P_W is a program with the embedded watermark W, and ii) a watermark extractor function, $F_{extract}$, extracts the watermark W' from P_W with metadata M_v, and veri es the extracted watermark W' with

$$F_{extract}(P_W, M_v) = \begin{cases} W' & \text{if } W = W' \text{ (Valid)}; \\ 1 & \text{if } W \neq W' \text{ (Invalid)} \end{cases}$$

2.2 Requirements

As with previous work [14, 21, 23, 27, 42, 49, 67] on software watermarking, we informally de ne six key properties (metrics) to evaluate the e ectiveness of the watermarking scheme. Note that any watermarking system exhibits a trade-o between these metrics; a high capacity (data rate) implies low stealth and resilience.

- **Resiliency**: A watermark must be robust against varying corruption attempts: waterwark invalidation, tampering, addition or deletion. Moreover, even when a target software with the watermark has been altered, an ideal watermark scheme should maintain its validity or (at least) remain partially recoverable.
- **Spread**: An ideal watermark should be distributed all over a program to protect as many parts as possible. A well-distributed watermark o ers probabilistically better resiliency.
- **Credibility**: A watermark should be reliably recoverable for the proof of the authorship. A false positive case (i.e., extracting a watermark from software without a watermark) or false negative case (i.e., failing to extract a watermark from software with a watermark) should be minimal.
- **Capacity**: A watermarking algorithm should be able to convey a certain amount of information (i.e., data rate) within a target program. It is desirable to quantitatively compute the maximum length of the watermark that can be encoded inside the program.
- **E ciency** : A watermarking scheme should have a negligible impact on a target program in terms of performance and space overhead.
- **Imperceptibility** : A watermark should be stealthy (like invisible or inaudible data from video/audio les) enough not to be detected by an adversary. A program with the watermark must be indistinguishable from another without the one.

2.3 Threat Model

It is a common belief that, a determined adversary with a su cient amount of resources will eventually be able to defeat any watermarking systems. Hence, our objective is to develop a watermarking technique that substantially thwarts every reasonable e ort with feasible resources in practice, rather than building an unbreakable scheme. With this in mind, in this section, we describe

Figure 1: Motivational example. Each column represents instruction addresses, machine codes in bytes and disassembled instructions of the original program (left) and the one with a watermark (right).

a threat model with several assumptions, followed by a group of viable attacks.

Code Signing. A program can be digitally signed to prevent unauthorized changes [34]. Code signing involves with a cryptographic signing process using a public/private key pair that uniquely belongs to a program owner where the public key has been certi ed from the trusted third party. By signing code, an adversary's ability to damage a watermark is severely limited, because code modi cation is disallowed[1]. However, code signing does not provide traceability of the binary, which SoftMark aims to provide. Code signing is a common practice: most benign software products are signing codes, and even malware campaigns take advantage of it [37]. Hence, it is reasonable to assume that a software copyright owner intends to leverage both code signing and watermarking to better protection, however, we assume that a target program may or may not be digitally signed.

Attack Scenario. We assume an adversary who may i) perform code manipulation or transformation when code signing has not been applied, ii) have the knowledge of the way how a watermark can be embedded, iii) collusively possess multiple program instances where each of which contains a di erent watermark, and iv) have a watermark extractor without the details of a program. We assume that the attacker does not have access to both the master binary (i.e., metadata containing function locations) and bookkeeper (or ledger) that contains useful information pertaining to a watermark (e.g., key functions used for the watermark, as described in Ÿ4.3.3), because an adversary can create a new unauthorized watermark with those information. Note that protecting the master binary and the bookkeeper from leakage (e.g., insider threat [3, 9]) is out of scope.

[1]A stolen certi cate to sign a binary program after modi cations (e.g., Stuxnet [54]), is orthogonal to our approach, hence out of the scope.

Attack Types. A robust software watermarking scheme should be able to thwart di erent types of attacks under our threat model. We classify such threats into three major groups as follows.

Unauthorized Detection represents a risk that an attacker recognizes the presence of a watermark within a program. Hiding the presence of the watermark is of utmost importance, while developing a perfect detection-proof watermark is extremely dif cult due to unexpected side channels. This attack corresponds to the property of imperceptibility.

Illegal Corruption encompasses exhaustive attacks that aim to destroy a legitimate watermark by i) insertion (i.e., additive attack that attempts to implant another valid watermark), ii) deletion (i.e., subtractive attack that attempts to completely eliminate a valid watermark), iii) alteration (i.e., tampering attack that attempts to counterfeit part or full of a valid watermark), or iv) distortion (i.e., ambiguity attack that attempts to puzzle a detector by applying semantic-preserving code transformations on a target program). These attacks correspond to the properties of credibility, resiliency and spread.

Collusive Attack aims to identify the location of a watermark by comparing multiple instances wherein di erent watermarking ngerprints have been embedded. A successful collusive attack leads to the location of or speci c pattern (rule) of a watermark, without using a legitimate watermark extractor.

3 DEMONSTRATIVE EXAMPLE

Since SoftMark inserts a watermark via the order of functions (Ÿ4), a major threat arises from attacks leveraging sematic-preserving code transformation. This section demonstrates watermark embedding and extraction of SoftMark with an example, focusing on the di culty of the transformation even with full accuracy.

Target Program. The original program in Figure 1 (a) has three functions that contain various code constructs including direct

call/jump (e.g., 0x400688 and 0x4007A0) and indirect call/jump (e.g., 0x400813 and 0x400898) instructions. SoftMark would select a set of functions from all function candidates for reordering.

Watermark with a Function Order. Given the three functions in the example, six (= 3!) possible orders can represent up to two bits as in Table 1. In principle, an individual watermark has a one-to-one mapping with a particular order of selected functions (e.g., 11_2 can be encoded at the order F_3 F_1 F_2 in Figure 1 (b).

Table 1: Function Order and Watermark Mapping.

Function Order	F_1 F_3 F_2	F_2 F_1 F_3	F_2 F_3 F_1	F_3 F_1 F_2
Watermark Value	00_2	01_2	10_2	11_2

Watermark Embedding. In our scheme, inserting a watermark essentially means generating a variant of the original program with relocated functions where the order of functions representing the watermark. Such code transformation inherently involves with a vast number of updating instructions such as immediate operands. Going back to the example, direct call/jump instructions (e.g., E8 or E9 in x86) can be trivially updated by recalculating the immediate operands ( ). However, indirect call/jump instructions require reference updates in a jump table that resides in the data region ( ), which is non-trivial. A runtime error would occur if any exercising code pointer update were failed. Moreover, successful function relocation requires a clear function boundary () because it may break the original semantics otherwise.

To exemplify, the values at 0x400998 in (b) in the data section point to the call instructions at 0x4006FC and 0x400706 that have been relocated from 0x40089D and 0x4008A7. If any of those addresses has not been updated properly, the program would cause a runtime error. Similarly, the function pointers at 0x400210 in (b) that point to the function 2 and 3 must be appropriately updated according to the functions' new addresses. This imposes a non-trivial challenge to those who attempt to compromise our watermark by relocating functions. Moreover, another challenge is to identify an accurate boundary between code pointers and raw data. In this example, the values in purple at 0x402040 in (a) and 0x400220 in (b) are scalar data (i.e., not code pointers) between two jump tables, which are indistinguishable from surrounding code pointers. To launch a successful attack, an adversary should be able to di erentiate the boundary of code and data, which is undecidable. SoftMark takes advantage of a special compilation toolchain [35] that produces metadata for reliable static binary instrumentation (e.g., function boundary and jump table), and record unique information for a watermark when generating a mutation corresponding to the watermark. Note that the metadata produced by [35] is critical and kept secret from adversaries (Details in Ÿ7).

Watermark Extraction. It is straightforward to extract an embedded watermark. We can identify the order of functions with the recorded information, followed by decoding a watermark according to Table 1. Ensuring the integrity of a target binary, we discuss the case when the binary has been compromised in Ÿ4.4.

4 SOFTMARK DESIGN

This section describes the design of SoftMark that satis es the requirements (Ÿ2.2) against various attacks (Ÿ2.3) when embedding and extracting a watermark.

4.1 Overview

Figure 2 depicts a work ow of SoftMark . First, we employ a special compiler toolchain [35] to compile a given program from the source code. During the compilation, the toolchain generates metadata (e.g., locations of functions) for reliable static binary instrumentation, required for our watermarking embedding. We call the pair of the binary and metadata a master binary. Second, we analyze the binary and choose reorderable function candidates that can represent k bits of data with di erent orders of the n functions. Then, we generate a variant of the target program with a unique ngerprint via reordering of n functions. We also record the ngerprint and its associated identi er in a ledger (accessible merely by a product owner). Third, we extract the watermark from a binary by identifying the function order. Finally, a user associated with the extracted watermark is identi ed by looking it up the ledger.

4.2 Bene ts of Our Approach

The bene t of a static approach, including SoftMark , is two-fold: i) inexpensive; it can be easily adopted in large-scale applications at a low cost, ii) robust; a dynamic approach relatively su ers from watermark corruption as reversing techniques advance.

Advantages over Existing Techniques. We aim to mitigate previous drawbacks to meet the requirements of software watermarking (Ÿ2.2). Our function-reordering-based watermark approach offers the following three advantages. First, reordering functions is a semantic preserving transformation; that is, watermark insertion does not a ect the original program's semantic because SoftMark does not introduce any additional code, blocks or subroutines to a target program. While the relocated functions may change cache behaviors at runtime, our assessment demonstrates that its impact on the performance overhead is negligible (Ÿ6.5.2). Second, introducing no supplementary structure gives a relatively lower chance for attackers to recognize the presence of a watermark with a statistical analysis or inference. One conceivable scenario is a collusive attack that acquires multiple instances with di erent watermarks, which may unveil the presence of a watermark (i.e., by identifying the locations of the same functions between the instances). Nonetheless, our watermark stays resilient against any attempt of watermark extraction (Ÿ4.4) because the mapping information between a watermark and an order is still concealed in a private ledger. Third, the number of reorderable functions can reach up to an increasingly large number of encodings (i.e., n! with n functions); e.g., 10 di erent functions can produce millions of permutations, o ering a high data-rate encoding as the size of an application (and typically the number of functions) increases.

Existing Reordering-based Approaches. Reordering-based techniques are the closest existing approaches of SoftMark . However, unlike SoftMark , they su er from three major limitations. First, they are perceptive; Myles et al. [42]'s approach could be easily detectable because its implementation relies on inserting a

Figure 2: Overview of SoftMark 's Work ow. A software product owner prepares a master binary with metadata, and analyze it beforehand. A watermark is embedded with function reordering from pre-selected functions. Watermarking extraction veri es the watermark once identi cation of functions and their orders is complete.

Table 2: Number of functions and the size of representable bits accordingly. In our experiment, we insert a 256 bits long watermark that requires at least 58 functions for reordering, which can be adjustable according to a policy.

Functions	Representable bits	Bits	Required functions	Applicable examples
10	21	64	21	26 (433.milc etc.)
20	61	128	35	26 (433.milc etc.)
30	107	192	47	26 (433.milc etc.)
50	214	256	58	26 (433.milc etc.)
100	524	512	99	48 (482.sphinx3 etc.)
300	2041	1024	171	256 (hmmer etc.)
500	3767	2048	301	1911 (ygen, etc.)
1000	8529	4096	537	1700 (perlbench etc.)

Figure 3: The number of representable bits is proportional to the number of functions.

large number of GOTO statements to maintain the original control ow. Second, they are forgeable; rearranging a structure can be accomplished with a trivial e ort [52, 53, 55]. Third, they are fragile; watermarks were not resilient to arbitrary modi cations at the instruction level [28].

4.3 Watermark Embedding

In this section, we develop various techniques used in SoftMark to enhance the e ectiveness of watermark embedding.

4.3.1 Encoding Size. SoftMark 's watermark is encoded as the location information of functions in the target program. In other words, a particular order of a set of functions determines a certain watermark value. For example, consider a simple executable with three user-de ned functions.[2] The number of possible orders among the three functions is $3! = 6$, and it can represent up to 2 bits of a watermark ($2^2 < 3! < 2^3$); e.g., $00_2, 01_2, 10_2$, and 11_2. As the number of permutation of n distinct objects allows for $n!$ di erent ways of function reordering in total, the number of possible watermarking bits can be asymptotically computed according to the Stirling's formula [63] (approximation for factorials) as in Equation 1.

$$\log_2(n!) \approx \log_2\left(\sqrt{2\pi n}\left(\frac{n}{e}\right)^n\right) \text{ where } n > 0 \quad (1)$$

Table 2 summarizes the number of functions and the size of required bits that represents a watermark accordingly, which can be determined by a watermarking policy. Figure 3 illustrates the linear relationship between the number of representable bits and the number of functions.

4.3.2 Embedding Strategies. As watermark embedding process must support a deterministic extraction process without ambiguity. Besides, it should o er both credibility and robustness against watermark corruption attempts. We develop strategies for deterministic, credible, and robust watermark embedding and extraction.

Unique Function Candidates. A program may contain multiple functions that have indistinguishable (i.e., identical) binary code. We do not use such functions in our watermark embedding because our extractor cannot distinguish the locations of the functions (as well as the order between them). As an example, consider that a program consists of four functions F_1, F_2, F_3, and F_4 where the last two are indistinguishable, then a watermark with an order of F_3, F_1, F_4, F_2 could be interpreted as F_4, F_1, F_3, F_2. Such multiple interpretations raise a false positive case, violating the property of credibility. It is noteworthy mentioning that the selected functions in Table 2 should be all unique. Toward uniqueness of candidate functions, we de ne a unique function as the one that comprises a unique combination of basic blocks where each block has a unique sequence of instruction mnemonics.[3]

Exclusion of Small Functions. We empirically discover that approximately 76% of non-unique functions, on average, consists of a single basic block or even a single instruction in the programs for our evaluation (Figure 4). Note that, in such small functions, it is not rare that two functions have an identical sequence of instructions. Thus, we intentionally rule out those small functions from the candidate for a watermark. The column Small in Table 4 shows the number of small functions.

Desirable Function Set. Once we have the list of candidates to choose from, we carefully pick functions that make static binary

[2] We rule out linker-inserted functions during compilation because di erent linkers may introduce di erent number (and kind) of additional subroutines such as CRT.

[3] We do not consider operands (e.g., immediates) because they should be updated to obey the original ow while displacing a function.

Figure 4: Ratio of small functions (i.e., containing a single basic block) of all non-unique functions. It ranges from 60% to 95% with an average of 76% (dotted line).

instrumentation challenging. When we select a candidate, we prefer a function that has a trampoline containing code pointers (i.e., indirect jumps or calls) because displacing such a function raises the bar. Speci cally, an attacker needs to update a data region that embodies both code pointers and scalar data values for successful binary instrumentation. The column iCFT (indirect Control Flow Transfer) in Table 4 shows the number of desirable functions containing indirect branches.

Basic Block Reordering. We apply a basic block transposition within a function against collusive attacks. Note that this does not a ect the identi cation of unique function candidates because the order of basic blocks is not the factor of function uniqueness.

Multiple Watermarks. By design, we allow a software owner to be able to insert a watermark multiple times so that one could recover at least one of them when an active adversary attempts to tamper with the watermark by partial function relocation. We leave it as a hyperparameter, k, and empirically set it up as $k = 3$.

Non-candidate Functions. We randomly scatter all other functions that are not part of the functions for a watermark, increasing resiliency against collusive attacks. The order of scattered functions is deterministic for each watermark.

4.3.3 Ledger (Bookkeeper). Along with watermark embedding, an owner must maintain a private ledger (i.e., bookkeeper) that holds the list of functions and their locations (Ÿ2.3), the property of each function (e.g., index, indirect call invocation), and the pattern of basic blocks for further watermark extraction. In particular, a basic block can be recognized with two means: i) regular expression of byte values for a quick search and ii) the sequence of opcode mnemonics and sizes after disassembly for deep investigation (Ÿ5). Note that the ledger should remain undisclosed so that adversaries cannot acquire it.

4.4 Watermark Extraction

Extracting a watermark is a reversing process of the embedding process. We assume two possible scenarios in watermark extraction: a case that a target binary stays intact, and another case that the binary has been corrupted (or altered). Although it su ces to say that the extraction is a success or failure, our goal is either to precisely obtain a genuine watermark (for the former case), or to partially recover it as a best-e ort service (for the latter case).

4.4.1 Function Identification. As we deal with a stripped binary, it is required to recognize function boundaries. First, we con rm basic block patterns from the list of selected functions for a watermark, followed by seeking all blocks with those patterns. Then, a certain function can be identi ed if a set of basic blocks is present in the function.

4.4.2 Extraction from Unmodified Binary. As with a secret ledger, a watermark extraction is straightforward and precise without vagueness because we solely use unique functions for watermark embedding.

4.4.3 Extraction from Modified Binary. If a target binary has been corrupted, the information in a ledger for the master binary does not match, raising a failure of both watermark detection and extraction. In such a case, SoftMark attempts to extract a watermark from unmodi ed code parts because the authenticity of a corrupted function and its location cannot be trusted. As SoftMark can insert multiple watermarks across a wide range of original code, a partial extraction of those may su ciently reveal the ngerprint.

5 IMPLEMENTATION

This section brie y describes SoftMark implementation. Our prototype currently supports ELF executables for the x86-64 platform on Linux. We leverage the CCFR [35] toolchain based on a modi ed LLVM (v3.9.0) and gold linker (v.2.27) to relocate functions and basic blocks. We developed our binary metadata analysis tool in Python, which takes a master binary that contains metadata for watermarking as an input. We use the pyelftools [7] for parsing an ELF format and the capstone [24] library as a disassembly engine.

Basic Block Pattern Search. We implemented two di erent techniques for seeking basic blocks as part of a function identi cation phase in Figure 2. By default, SoftMark discovers blocks in a ledger with a pattern using regular expressions under the assumption that a given binary has not been compromised, which allows for a quick search without a hassle. However, in case that the given binary has been modi ed (e.g., code transformation), it requires a deep search with a full disassembly process by matching instruction opcode and size (for recognizing the boundaries of blocks and functions). There is a coincidental case to take into account with the deep search where a consecutive functions have overlapping blocks. For example, if two functions of F_1 and F_2 share blocks like $F_1 = (B_1, B_2, B_1)$ and $F_2 = (B_2, B_1, B_3)$, it would be problematic when SoftMark mistakenly recognizes F_1 that comes from two blocks, say, (B_1) from F_1 and (B_2, B_1) from F_2. Although we empirically observe that this rarely happens, we intentionally avoid such cases by re-embedding a watermark. Table 4 illustrates the comparison of embedding and extraction time between the regular-expression and disassembly-oriented implementation, whose di erence is as orders of magnitude as large. Thus, it is recommended to try a deep search with full disassembly when only needed for further investigation.

6 EVALUATION

We evaluate SoftMark on a 64-bit Ubuntu 18.04 system equipped with Intel(R) Core—i7-6700 3.40 GHz and 8GB RAM.

Corpus. We collect 26 binaries for SoftMark evaluation from various dataset including eight programs from SPEC CPU2017 [11], samples of Binutils v2.27 from GNU Project [25], and seven utilities of our choice (e.g., pu y /pscp/ps p v0.75 [58], vs pd v2.3.4 [60], ctags v5.9.0 [19], and ligh pd v1.4.32 [38]). We have excluded applications that do not contain su cient function candidates (i.e., at least 58 functions or above for embdding 256-bit identi er in our experiment) such as bzip2, mcf, and specrand.

6.1 Resiliency

As SoftMark operates on watermark embedding and extraction solely at a binary level, we consider possible corruption attacks focusing on machine code.

Distortion to Complicate Watermark Extractor. As described in Ÿ2.3, our SoftMark scheme may be susceptible to a distortion attack with semantic-preserving code transformations [36, 45] because such an attack impedes the proof of original program's authenticity during a watermark extraction process. Note that while those techniques do not change the location of a function, they may challenge a watermarking extraction process by breaking the integrity of code. Our disassembly-oriented basic block search is robust to operands distortion (e.g., register reordering and assignment) but opcode distraction (e.g., instruction substitution and reordering) may lower a survival rate (See Ÿ7 in detail). Note that the aforementioned attacks can be simply prevented with code signing.

Function Relocation. As SoftMark relies on function reordering, it is susceptible to attacks that can relocate functions. Table 3 shows precision, recall, and F1 score of the function boundary detection technique implemented in the state-of-the-art disassembler, IDA Pro v7.2 [29]. The precision, recall, and F1 scores are 0.823, 0.521, and 0.629, on average, respectively. The results show that SoftMark successfully imposes signi cant challenges to adversaries in practice. We take ground truths from unstripped binaries with debugging information.

Other Obfuscation Techniques. We review varying obfuscation techniques o ered by Sandmark [13] and others [13, 16, 31, 56, 62] to see whether they can be used to corrupt (i.e., attack) SoftMark. Since most approaches in Sandmark leverage obfuscation or optimization at a source code or Java Bytecode level to generate a valid watermark, which are not applicable to our context (i.e., binary level), we assess techniques that can be applied at a machine code level as following:

Code Insertion : This attack aims to insert additional instructions (e.g., code displacement [35]) , basic blocks, or functions. It will inevitably change the size of a target binary, rendering an attempt of distortion detectable. This partially thwarts our scheme by hindering function identi cation properly.

Constant Modi cation : An attacker may corrupt constant values in a data region. However, it does not a ect SoftMark 's watermark, as our watermark scheme merely relies on the order of functions.

Basic Block Reordering : An attacker may reorder basic blocks (as there are basic block reordering watermarking techniques) in a function. SoftMark is resilient to this type of attack since we recognize a target function with unique basic blocks that are agnostic to their orderings.

Table 3: Precision, recall and F1 scores of function boundary detection with IDA Pro [29]. Identifying clear boundaries from stripped binaries using a state-of-the-art reversing tool is insu cient for binary instrumentation.

	stripped binary		
Program	Precision	Recall	F1 Score
400.perlbench	0.828	0.611	0.703
403.gcc	0.804	0.576	0.671
433.milc	0.859	0.653	0.742
445.gobmk	0.830	0.245	0.379
456.hmmer	0.880	0.505	0.642
458.sjeng	0.777	0.626	0.693
464.h264ref	0.856	0.693	0.755
482.sphinx3	0.843	0.594	0.697
addr2line	0.827	0.401	0.540
ar	0.832	0.414	0.553
bfdtest1	0.816	0.396	0.533
cxxfilt	0.836	0.406	0.547
nm-new	0.843	0.421	0.562
objcopy	0.826	0.464	0.594
objdump	0.818	0.475	0.601
ranlib	0.798	0.400	0.533
size	0.840	0.406	0.547
strings	0.803	0.393	0.527
strip-new	0.831	0.470	0.600
ctags	0.845	0.576	0.685
ligh pd	0.799	0.735	0.766
vs pd	0.854	0.791	0.821
pscp	0.802	0.621	0.700
ps p	0.791	0.622	0.697
pu ygen	0.780	0.515	0.620
cgtest	0.789	0.524	0.630

Figure 5: Boxplot of embedding a watermark ten times. The results show that embedding time is quite consistent (i.e., low variance) for all programs.

Branch Instruction Modi cation : Modifying branch instructions may also complicate our scheme because it can alter an instruction opcode as well as a control ow change.

6.2 Spread

We evaluate how pre-selected functions are spread throughout a binary in SoftMark . Well spread functions enable a watermark to be robust against random modi cation of a binary; in other words,

Table 4: Experimental Evaluation Dataset and Results. Pre Anal., Em., Ex., iCFT and O/H represent pre-analysis, embedding, extraction, indirect control ow transfer and a performance overhead, respectively.

Program Name	Layout		Functions Candidates					Time with a regular expression (sec)						Time with a disassembly (sec)						O/H
	Func	BBLs	Small	Unique	Not Unique	iCFT	iCFT Size Ratio (%)	Pre-Anal.	Em.	Ex. W1	Ex. W2	Ex. W3	Avg.	Pre-Anal.	Em.	Ex. W1	Ex W2	Ex. W3	Avg.	
400.perlbench	1,660	46,682	622	895	143	153	23%	60	15	2.1	1.6	1.8	1.8	27,426	16	3,372	309	1,432	1,704	-1.9%
403.gcc	4,329	118,085	1,924	2,206	199	515	30%	346	101	6.5	5.8	4.2	5.5	200,270	101	2,471	2,445	2,125	2,347	0.
433.milc	235	2,586	141	88	6	7	2%	2.2	0.8	0.3	-	-	0.3	12	0.8	6.8	-	-	6.8	-7.2%
445.gobmk	2,478	25,021	1,374	857	247	36	2%	51	34	0.7	1.0	0.8	0.8	6,282	34	352	139	133	208	-1.7%
456.hmmer	470	10,154	221	237	12	30	14%	5.9	1.7	0.6	0.5	0.9	0.7	437	1.7	38	57	46	47	1.1%
458.sjeng	132	4,475	51	78	3	16	26%	2.3	0.9	0.6	-	-	0.6	25	0.9	20	-	-	20	0.6%
464.h264ref	518	13,986	227	259	32	36	9%	13	3.5	1.2	1.2	1.1	1.2	1,358	3.5	94	75	64	78	0.2%
482.sphinx3	318	5,350	152	155	11	7	4%	3.4	1.5	0.3	0.4	-	0.3	102	1.5	15	16	-	15	0.5%
addr2line	1,459	35,718	651	704	104	297	39%	39	11	0.7	0.9	2.2	1.3	13,534	11	804	630	846	760	-
ar	1,522	36,589	676	739	107	305	39%	40	11	1.2	0.9	0.6	0.9	14,982	12	797	812	782	797	-
bfdtest1	1,429	35,293	641	686	102	294	39%	39	10	0.8	0.6	1.9	1.1	11,459	11	645	662	646	651	-
cxxfilt	1,458	35,674	649	705	104	297	39%	39	12	0.7	0.9	2.1	1.2	13,045	12	735	653	865	751	-
nm-new	1,483	36,099	654	721	108	304	39%	40	11	1.1	0.6	1.1	0.9	12,760	12	699	639	640	659	-
objcopy	1,694	43,505	720	867	107	329	39%	58	14	3.6	4.0	1.1	2.9	28,679	15	822	1,044	987	951	-
objdump	1,888	49,149	780	988	120	369	24%	87	23	4.3	0.9	4.5	3.2	37,856	23	901	1,521	1,243	1,222	-
ranlib	1,522	36,589	676	739	107	305	39%	40	11	1.2	0.9	0.6	0.9	12,949	12	365	494	457	439	-
size	1,464	35,734	652	708	104	298	39%	40	11	0.9	0.5	1.7	1.0	11,450	11	632	664	631	642	-
strings	1,460	35,760	651	705	104	297	39%	39	11	0.8	0.6	2.2	1.2	13,972	11	701	656	669	675	-
strip-new	1,694	43,505	780	867	47	329	39%	64	15	3.6	4.1	1.1	2.9	25,440	15	1,124	1,131	1,096	1,117	-
ctags	2,886	42,717	1,325	1,150	411	225	16%	19	19	0.9	1.0	0.7	0.9	26,012	20	539	455	322	439	-
ligh pd	291	4,244	132	136	23	28	22%	1.4	1.2	0.5	0.1	-	0.3	89	1.2	12	20	-	16	-
vs pd	510	3,474	335	143	32	9	3%	1.3	1.1	0.2	0.1	-	0.2	69	1.1	4.1	5.3	-	4.7	-
pscp	1,636	14,585	900	664	72	228	24%	15	4.2	0.5	0.3	0.4	0.4	2,304	4.3	116	138	149	134	-
ps p	1,653	14,716	906	671	76	219	22%	15	4.3	0.5	0.3	0.4	0.4	2,319	4.3	117	142	153	137	-
pu ygen	881	7,167	512	328	41	50	13%	2.4	2.6	0.5	0.3	0.2	0.3	555	2.8	41	50	29	40	-
cgtest	891	7,336	514	335	42	50	12%	6.4	1.8	0.6	0.2	0.3	0.4	310	1.9	35	46	32	38	-

a successful attack requires a wider range of compromising code under our scheme(e.g., altering instructions or reordering functions). Table 5 shows the probability of a successful attack by the size of a watermark. Simply put, a watermark would be corrupted (i.e., failed to be extracted) when an adversary randomly chooses and alters a set of functions via distortion or reordering. Note that 433.milcand 458.sjengdo not have the results for512bits due to the lack of function candidates to represent those bits. Figure 6 depicts a probability curve by the number of functions and the representable bits. In a nutshell, the higher data rate increases the attack probability whereas the larger number of functions decreases it. Based on our experiment, we recommend to use at least 1,500 functions with a 256-bit watermark for reasonable robustness.

Table 5: Attack probabilities by the size of a watermark. The number of selected functions for a watermark in 400.perlbench is 58out of 1;660; that is, the probability of choosing the exact set of those functions is 58 1;660= 0:0349

	Attack probability			
Name	64 bits	128 bits	256 bits	512 bits
400.perlbench	0.0127	0.0211	0.0349	0.0596
403.gcc	0.0049	0.0081	0.0134	0.0229
433.milc	0.0894	0.1489	0.2468	-
445.gobmk	0.0085	0.0141	0.0234	0.0400
456.hmmer	0.0447	0.0745	0.1234	0.2106
458.sjeng	0.1591	0.2652	0.4394	-
464.h264ref	0.0405	0.0676	0.1120	0.1911
482.sphinx3	0.0660	0.1101	0.1824	0.3113
addr2line	0.0144	0.0240	0.0398	0.0679
ar	0.0138	0.0230	0.0381	0.0650
bfdtest1	0.0147	0.0245	0.0406	0.0693
cxxfilt	0.0144	0.0240	0.0398	0.0679
nm-new	0.0142	0.0236	0.0391	0.0668
objcopy	0.0124	0.0207	0.0342	0.0584
objdump	0.0111	0.0185	0.0307	0.0524
ranlib	0.0138	0.0230	0.0381	0.0650
size	0.0143	0.0239	0.0396	0.0676
strings	0.0144	0.0240	0.0397	0.0678
strip-new	0.0124	0.0207	0.0342	0.0584
ctags	0.0073	0.0121	0.0201	0.0343
ligh pd	0.0722	0.1203	0.1993	0.3402
vs pd	0.0412	0.0686	0.1137	0.1941
pscp	0.0128	0.0214	0.0355	0.0605
ps p	0.0127	0.0212	0.0351	0.0599
pu ygen	0.0238	0.0397	0.0658	0.1124
cgtest	0.0236	0.0393	0.0651	0.1111

6.3 Credibility

A watermark is embedded based on a one-to-one mapping relationship of function order information. HenceSoftMark can precisely identify the watermark as long as all functions are unique and successfully extracted. Note that we exclude functions that may cause a false positive case as described in Ÿ4.3.2. Recognizing the functions can be complete with a basic block pattern search described in Ÿ5.

6.4 Capacity

We compare our approach with the one from Davidson et al.[22] that is based on basic block reordering in terms of capacity. As shown in Table 6, the data rate ofSoftMark is signi cantly higher than that of the Davidson's approach (up to15times for482.sphinx3). This is because our approach depends on the number of possible

Figure 6: Probability curve for successful attacks, which indicates that the higher number of functions or the lower capacity of a watermark, the lower probability of the attacks.

function candidates, in contrast, Davidson's approach predominately relies on the maximum number of basic blocks within a function. The downside of the latter approach arises from which the largest number of basic blocks has nothing to do with the size of a program, which may not be su cient to represent a watermark. For example, 403.gcc has 2.5 times more functions than 400.perbench, however, the maximum representable bits is rather 20% smaller. We discuss the capacities of other watermarking techniques that cannot be directly compared with SoftMark in Ÿ7.

Table 6: Comparison of capacity (i.e., maximum number of representable bits) between SoftMark and Davidson's approach [22] that relies on the largest block size in a function.

Program Name	Size (KB)	SoftMark		Davidson-Myhrvold	
		Functions	Bits	Basic Blocks	Bits
400.perlbench	1,423	895	7,491	683	5,451
403.gcc	3,728	2,206	21,326	534	4,073
433.milc	148	88	446	20	61
445.gobmk	3,923	857	7,119	135	765
456.hmmer	339	237	1,532	46	191
458.sjeng	156	78	382	74	357
464.h264ref	685	259	1,708	160	945
482.sphinx3	210	155	909	20	61
addr2line	1,180	704	5,649	142	815
ar	1,213	739	5,982	142	815
bfdtest1	1,165	686	5,479	142	815
cxxfilt	1,179	705	5,659	142	815
nm-new	1,195	721	5,810	142	815
objcopy	1,410	867	7,217	181	1,101
objdump	2,474	988	8,409	186	1,139
ranlib	1,213	739	5,982	142	815
size	1,180	708	5,687	142	815
strings	1,180	705	5,659	142	815
strip-new	1,410	867	7,217	181	1,101
ctags	1,495	1,150	10,039	178	1,078
ligh pd	195	136	772	85	426
vs pd	118	143	822	87	439
pscp	713	664	5,273	99	518
ps p	722	671	5,338	99	518
pu ygen	391	328	2,273	144	829
cgtest	405	335	2,332	141	808

6.5 E ciency

6.5.1 Size Overhead. The size of a watermark-inserted binary stays identical because SoftMark does not introduce additional structures such as codes, blocks or subroutines to a target program (Ÿ4.3). We con rmed that each binary with a watermark for evaluation does not increase a code size.

6.5.2 Performance Overhead. The rightmost column in Table 4 shows the performance overheads of SPEC CPU2006 binaries after embedding a watermark with SoftMark. For each binary, we measured the overall CPU user time for the completion of all internal tests by taking the average time across ve runs, using both the original and its corresponding variant with a watermark. The largest overhead is reported with 456.hmmer, 1.1%, which is negligible. Interestingly, the performance of 400.perlbench, 433.milc, 445.gobmk, and 458.sjeng demonstrates slightly better than their original (master) binaries. We attribute those speedups in better caching behavior from a code region due to di erent code localities after function relocations, which aligns with the results from [35].

6.5.3 E iciency of Embedding and Extraction. We evaluate the eciency of our watermarking embedding and extraction process. Recall that we operate two di erent modes for a watermarking extraction depending on the assumption of a binary status: i) unmodi ed (identical) and ii) modi ed (compromised). Both cases refer a bookkeeper to identify the location of every function, but SoftMark carries out a block search di erently; the former employs a regular expression for performance where the latter employs a disassembly for deep binary inspection to recognize code alteration.

Unmodi ed Binaries. A default watermarking extraction with a regular expression in Table 4 shows pre-analysis, embedding and extraction of three di erent watermarks. A pre-analysis step examines the property of a function such as its uniqueness and indirect branches within. 403.gcc takes the longest time; that is 346, 97, and 5,5 seconds for pre-analysis, embedding, and extraction, respectively. Except for three programs (403.gcc, 445.gobmk, objdump), an embedding process takes less than 20 seconds, which is reasonable in practice. An extraction process takes up to 55 seconds where most of cases can be done within a few seconds.

Modi ed Binaries. Watermarking extraction with a disassembly in Table 4 shows that it takes longer time than handling a unmodied binary. This is mainly due to a substantial analysis to investigate potential attacks such as code transformation. A pre-analysis time varies, ranging from 12 seconds for 433.milc to 55.6 hours (220,270 seconds) for 403.gcc, depending on the size of a program. However, it is a one-time processing per each binary. The duration of embedding time is quite close to that of a unmodi ed binary case. An extraction process also takes longer than embedding, ranging from 5 seconds (vs pd) to 39 miniutes (gcc).

Di erent Values. We test three watermark values (Value #1-#3) by running inserting and extracting them 10 times. Note that each value can be generated by a software vendor as a secret identi er. Figure 7 and Table 7 depict the results of four representative programs' results by the number of functions including 456.hmmer (237 functions; small), pu ygen (328 functions; small), nm-new (721

Figure 7: Duration of embedding and extraction in seconds depending on inserting three di erent watermark values across four binaries. We empirically con rmed that the computational resources are agnostic to the watermark values.

Figure 8: Duration of embedding and extraction in seconds depending on the size of a watermark (i.e., 64, 128, and 256 bits) across four binaries. We con rmed that it takes longer time that is closely proportional to the size of the watermark on average.

Table 7: Di erences in embedding time according to watermark values and size changes (related to Figure 7, 8). Embedding time only shows a di erence of less than 1 second on the alteration of a watermark value or size.

	Embedding Value (256 bits)			Embedding Bits		
Name	Value #1	Value #2	Value #3	64	128	256
456.hmmer	1.8	1.8	1.8	1.7	2.6	1.8
nm-new	10.0	10.4	10.2	10.3	10.2	10.4
objdump	21.5	21.8	21.5	22.5	22.3	21.7
pu ygen	2.1	2.1	2.1	2.1	2.2	2.1

functions; medium), and objdump (988 functions; large). Empirically, we con rmed that a watermark value does not have large variations for embedding and extraction (Table 7).

Di erent Sizes. As shown in Table 7, embedding time is consistent on the di erent sizes of watermarking. However, we observe that the extraction of a watermark with a high-data rate takes longer time than that with a low rate. Interestingly, there are a few cases where extracting a 64-bit watermark is slower than 128-bit. It turns out that the functions in a 64-bit watermark contains more basic blocks. This is because our extraction performance depends on the complexity of a function for function identi cation. In general, embedding a larger watermark requires more functions, increasing the chance of dealing with a complicated function.

6.6 Imperceptibility

As discussed in Ÿ4, SoftMark does not add any explicit structure to a binary, which remains little information behind. Hence, inspecting a single mutation would not reveal any sign of a watermark. However, the parties in collusion who are aware of the principle of our watermark scheme (i.e., reordering-based) may learn the presence of a watermark by collecting multiple instances. Although SoftMark is equipped with several techniques to complicate function recognition (e.g., by basic block reordering within a function when generating a variant), a collusive attack can eventually thwart imperceptibility.

7 DISCUSSION AND LIMITATION

This section discusses future research and limitations of our work.

Binary Packing. Every watermark scheme on a binary code would be a ected by binary packing because it involves with a widely destructive process for a code region. When a program with a watermark is packed, it is required to unpack/dump the program on memory at runtime, followed by performing watermark veri cation on top of the dumped code. If unpacking were failed (e.g., a customize packer), SoftMark cannot reveal a watermark.

Semantic-preserving Code Transformation. ORP [45] proposes four in-place code randomization techniques without maintaining the size of a program: instruction substitution where an adversary replaces an instruction with another that is semantically equivalent, instruction reordering within a basic block by pre-computing possible orderings of given instructions, register reordering with the pair registers on the stack for a function prologue (e.g., push) and epilogue (e.g., pop), and register reassignment by reallocating swappable registers with pre-computing live regions in a function. As stated in Ÿ6, our scheme (even with multiple watermarks) cannot fully thwart such semantic-preserving code transformation attacks by altering opcodes (and), which we leave part of our future work. An instruction displacement technique [36] demonstrates another possible code transformation but it alters a program control ow with a jmp and its size, which can be easily perceptible. Egalito [40] allows arbitrary modi cation at a binary level with a layout-agnostic intermediate representation, however, it only supports a position-independent executable (PIE) for rewriting a binary. Besides, SoftMark can still judge a given program that a watermark has been corrupted when it may have failed the watermark extraction.

Collision with a Function Relocation. Note that even for an attacker who can reorder functions, the probability of nding a successful collision (i.e., legitimate entry) is extremely low without