Software Watermarking using Return-Oriented Programming

Haoyu Ma*
ma-haoyu@163.com

Kangjie Lu†
kjlu@gatech.edu

Xinjie Ma*,‡
xjma@smu.edu.sg

Haining Zhang*
guage_110@sohu.com

Chunfu Jia*,§
cfjia@nankai.edu.cn

Debin Gao‡
dbgao@smu.edu.sg

*College of Computer and Control Engineering, Nankai University, China
†School of Computer Science, Georgia Institute of Technology, USA
‡School of Information Systems, Singapore Management University, Singapore
§Information Security Evaluation Center of Civil Aviation, Civil Aviation University of China, China

ABSTRACT

We propose a novel dynamic software watermarking design based on Return-Oriented Programming (ROP). Our design formats watermarking code into well-crafted data arrangements that look like normal data but could be triggered to execute. Once triggered, the pre-constructed ROP execution will recover the hidden watermark message. The proposed ROP-based watermarking technique is more stealthy and resilient over existing techniques since the watermarking code is allocated dynamically into data region and therefore out of reach of attacks based on code analysis. Evaluations show that our design not only achieves satisfying stealth and resilience, but also causes significantly lower overhead to the watermarked program.

Categories and Subject Descriptors

K.5.1 [**Legal Aspects of Computing**]: Hardware/Software Protection – Proprietary rights

General Terms

Security

Keywords

Software watermarking, return-oriented programming, reverse engineering, code obfuscation

1. INTRODUCTION

Software theft and pirating have always been important concerns in software industry. In fighting against such intellectual property violations, software watermarking is considered a valuable tool. Like media watermarking, software watermarking embeds a secret message into a subject program which can be extracted to identify the copyright owner or authentic user of the program.

Existing software watermarking designs can be divided into static and dynamic watermarking [10]. The former embeds the watermark message directly in program's text, while the later hides it in program's runtime states or dynamic data. Since dynamic watermarking retrieves the hidden message by running along a specific path of the watermarked program and examining its specific behavior, it is usually considered a more reliable and secure solution [9].

Nevertheless, despite the progress made on dynamic watermarking, there are well-documented limitations with existing techniques [9]. First, existing solutions of dynamic watermarking introduce special data structures and instruction patterns which could be targeted by attackers to locate the hidden watermark. Second, code inserted by dynamic watermarking is usually rather independent from the other parts of the program, and thus can be suspicious. Finally, because previous designs, e.g., [8, 21, 23], use an external extractor to recover the watermark from program's execution recordings, sometimes the watermarked program needs to leave some information for the extractor to find the hidden watermark, which might also be exploited by attackers in undoing it. Existing research suggests that these problems be solved by integrating other protection techniques [23]; yet doing so also significantly decreases the efficiency of the watermarking technique [9].

In general, dynamic watermarking is to give the program a new execution path (which presents the watermark). However, with regular programming techniques, it's nearly impossible to also conceal the existence of this new execution path. In this paper, we present a innovative solution to this difficulty – a novel dynamic watermarking design based on *Return-Oriented Programming* (henceforth ROP), a well-known software exploit technique [4, 14, 16, 26, 27]. We show that, although initially proposed for malicious purposes, ROP can be applied in benign uses like software watermarking and works surprisingly well.

Our approach "assembles" the watermarking code out of a group of small, special instruction pieces picked from existing code, resulting an unexpected execution path that can only be chained with ROP. We also modify the subject program so that it prepares all other resources needed to chain

ASIA CCS'15, April 14–17, 2015, Singapore, Singapore

http://dx.doi.org/10.1145/2714576.2714582.

this watermarking path on-the-fly in its heap region. Only when triggered by the secret input will the program transfer its control to the hidden ROP path, which then extracts the embedded watermark. Our method ensures that the watermarked program does not have an explicit code stream that belongs exclusively to watermarking, so that when our watermarking module lurks it functionally doesn't exist, preventing it from being spotted by software analyses.

To our best knowledge, our approach is the first dynamic watermarking technique that performs execution of watermarking using solely instructions from other existing code modules. This not only keeps the watermarking code out of the scope of analysis tools, but also helps it survive various distortive attacks. Moreover, since ROP is initially developed to defeat protections against code injection exploits, our approach also involves no suspicious behavior like violation of W⊕X, DEP, or code signing. We present evaluations of program stealth and resilience in Section 4. Results show that our approach is able to make watermarking semantics untraceable by powerful static analysis tools, thus succeeds in concealing watermarking behaviors.

Another benefit of arranging the watermarking module in the form of ROP is better integration with the original program without creating suspicious data structures as in most existing work [11, 21]. We show that components of the watermark, which are inserted as ROP resources, can be spread out and blended with the program's other runtime data more easily and flexibly, minimizing suspicion from analyses that aim to locate them. Our evaluation demonstrates that our design achieves satisfying static stealth in merging the watermark into subject programs.

Last but not the least, instead of using an external extractor, our approach plants an ROP trigger inside the watermarked program to activate the ROP execution and to extract the watermark. This avoids leaking hints that could assist watermark recovering to third parties as in existing techniques [8, 11, 20, 21].

The rest of the paper is organized as follows. In Section 2 we introduce the background as well as existing research on aspects related to our design. The ideology and implementation of ROP-based watermarking is given in Section 3. Section 4 presents evaluations on the proposed solution along with comparisons with previous watermarking methods. We discuss applicability and compatibility issues of applying ROP in watermarking in Section 5. Section 6 concludes the paper.

2. RELATED WORK

2.1 Software watermarking

The goal of software watermarking is to embed an identifying message into a piece of software which could later be extracted to recognize the ownership or certain authentic information of the watermarked software.

Some schemes statically embed watermark into program's text by, e.g., relocating registers or modifying program's abstract semantics [12, 18]. Others focus on dynamic approaches in which the watermark is embedded in program's runtime behavior [8, 11, 19–21, 23, 25, 31].

Previous work on dynamic watermarking falls into two categories. Graph-based watermarking, first introduced by Collberg et al. [10], is one of the most well-understood software watermarking method [11, 23, 31, 34]. These schemes encode watermark messages into heap-allocated graph structures. Watermark extraction is done by examining graph structures built by the program with external watermark extraction routines and recognizing the graph that represents the watermark message. Another category of dynamic watermarking attempts to encode the watermark in special states of the program's control flow and extract the watermark by analyzing specific execution traces of the program, such as multi-thread behavior [21], conditional branching [8, 20], or value of opaque predicates [19].

A common characteristic of the above methods is that they require special components added into the program that serve solely for watermarking (e.g., data structures representing graph nodes [11,34], special thread components [21]). Many of them leave distinguishable features on regular running since the watermark extraction relies on external observation/examination to the program. For example, watermarking schemes based on thread behavior or dynamic execution path require locating and monitoring special thread components or certain branch instructions externally [9].

2.2 Return-oriented programming

Initially proposed by Shacham [27], return-oriented programming has become a major step in the advance of malicious code. While protection mechanisms like W⊕X and DEP are used by more and more operation systems to fight against code injection attacks, ROP provides a new way of exploitiation for arbitrary computation and bypasses the above protections since it does not inject any new code.

ROP was started on x86 architecture [5, 26] and later extended to many other platforms, e.g., the SPARC [4], ARM [16], etc. ROP attack is now not only fully automated, but able to be initiated during the executing of target program [15, 29]. Furthermore, ROP is also proven to be useful in compromising iOS applications [32] where code injection is not allowed.

2.3 Program steganography with ROP

Though introduced as an attacking technique, recently the argument on whether ROP can be used in a benign way has been brought to researchers' attention. In particular, RopSteg [17] was proposed for code protection that attempts to hide selected code portion of a program by executing their "unintended matches" located elsewhere. In the sense of hiding certain code blocks of a piece of software, RopSteg and our ROP-based software watermarking share similar purposes. However, RopSteg's design assumes a scenario where tampering with program code is not of the best interest of the attackers, which is certainly not true in software watermarking since removing a specific part of the program (that for watermarking) is exactly the purpose of attacks. When used directly for software watermarking, RopSteg has the following drawbacks.

First, RopSteg replaces the protected code portion with *ROP generator* and *ROP board* for jumping into and returning from the unintended matches. In watermarking this means that RopSteg still introduces new code of special pattern that only executes in watermark extraction. This could make the embedded watermark even easier to be located.

Second, RopSteg does not withstand distortive attacks in the sense that any simple program transformation applied to the RopStegged program would most likely destroy the

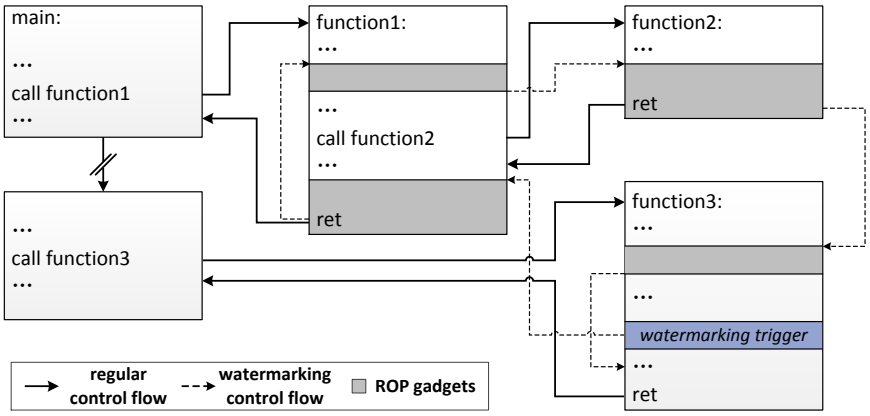


Figure 1: Overview of ROP-based watermarking

unintended matches of the hidden instructions. This makes it a fragile solution for watermark embedding.

3. ROP-BASED DYNAMIC SOFTWARE WATERMARKING

3.1 Threat model and assumptions

A dynamic watermarking design consists of a watermark *embedder* and a watermark *recognition protocol*. The watermark embedder, denoted as $E(\cdot,\cdot,\cdot)$, takes as input the subject program P, a watermark object ω, and an input setting $\delta \in \Delta$ (aka the "secret input"), and outputs the watermarked program $P' = E(P,\omega,\delta)$. The goal is to make $P'(\delta) = P(\delta) + \Omega$ at semantic level, where Ω is the executable form of ω, while preserving the semantics for other inputs, i.e., $P'(i) = P(i)\ \forall i \neq \delta$. The watermark recognition protocol $R(\cdot)$ extracts ω from P' when it runs with the secret input, i.e. $\omega = R(P'(\delta))$.

In general, an adversary may launch the following attacks on P':

Additive attack which turns P' into P'' in order to insert a bogus watermark υ, such that for an adversary-specified $\sigma \in \Delta$[1], $R(P''(\sigma)) = \upsilon$.

Subtractive/distortive attack which turns P' into P'' so that the authentic watermark ω is removed or compromised, i.e. $R(P''(\delta)) \neq \omega$.

The adversary is assumed to possess full control of P'. He could modify it using any semantic-preserving transformations, and observe and analyze its behavior both statically and dynamically. Nonetheless, the adversary is not al-mighty either. In practice, it is considered to be blind on a number of key information [10], namely:

1. Whether P' has been watermarked;
2. The original program P;
3. The secret input δ.

Our ROP-based watermarking inherits the above assumptions and considers the same general attack model. Furthermore, we extend the concept of the recognition protocol R to be more than just an algorithm or software analysis toolkit, but to also include the entire system environment setting in which the program is to be executed. The reason of such a definition will be explained in Section 5.

[1]Whether σ satisfies $\sigma \neq \delta$ is not important here.

3.2 Overview

The reason of ROP being so popular in malicious uses is that it formats its "code" in a form that is not to be run directly, but to help creating unexpected executions out of a program using its own instructions. Due to this special way of organizing executables, ROP also makes it very difficult for analysts to detect or understand the new "code" stream it creates. In other words, the execution path introduced by ROP is functionally "invisible". This could very well be a solution to building a dynamic software watermarking that, unlike existing ones, achieves minimum visibility while maintaining strong resilience.

It is important to first understand that an ROP "code" consists of 2 portions:

1. small pieces of instructions called "gadgets" that end with return or indirect call/jump, which are located somewhere in the existing executables; and
2. a carefully crafted bit string called the "payload" consisting of addresses of these gadgets as well as other variables they need in the execution.

ROP works by executing the gadgets with the payload strings, which means that for the new execution path it creates, nothing more than some extra data needs to be built. Therefore, the idea of ROP-based watermarking is to construct the actual watermarking code with ROP gadgets, while introducing extra code that builds a payload to control the gadgets. As briefly demonstrated in Figure 1, program modified in this way will transfer control to the selected gadgets when given the secret input[2] which takes over and presents the hidden watermark, and then return back to the program's normal routine. Since building ROP payload does not change the way in which the original program works, the extra code added by this method can always be executed during regular use of program instead of being kept idle and raising suspicions.

[2]These gadgets may be intended code pieces of the program or unintended instruction sequences in the middle of some code session (in case of CISC instruction set).

To give an example, suppose that we are to embed the watermark "007" into the target program by putting it into a 4-byte character string S (see Figure 2). In this example, payload construction involves preparing a sequence controlling the following gadgets and variables on the heap (shaded area in Figure 2):

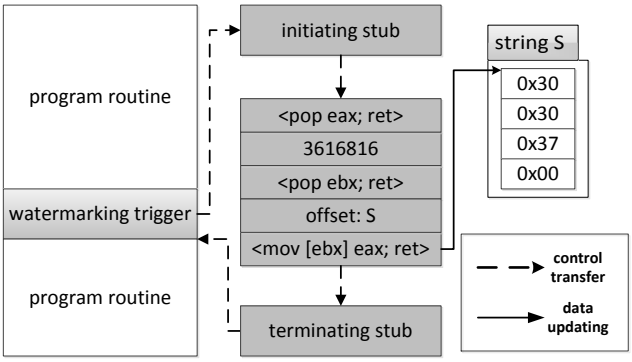


Figure 2: A simple example of ROP-based watermarking

- An initiating stub to store the current stack pointer and the address to return to, and to modify the stack pointer to point to the watermarking code;
- The watermarking code consisting of three gadgets `<pop eax; ret>`, `<pop ebx; ret>`, and `<mov [ebx], eax; ret>` which complete a memory writing chain;
- A constant 3616816 (corresponding to the ASCII code of string "007") and the address of string S which will be used by the memory writing gadgets;
- A terminating stub that recovers the original stack layout and goes back to regular program execution.

When the watermark extraction procedure is triggered, the embedded ROP first loads 3616816 and the address of S into register `eax` and `ebx`, respectively, then updates S to the watermark message "007"[3].

Typically, executing a well-designed ROP relies on injecting the payload on the stack in advance of turning to the gadgets, to ensure not only that the gadgets are given all operands they required but also that the indirect control instructions correctly find their successors so that gadgets are executed in the exact sequence as planned. Therefore, the way to construct and plant payload is a critical link of ROP. In malicious exploits, this is usually done by inputting external data to the program via some vulnerable routines, and the injected payload corrupts the original stack environment without concerning potential consequences (e.g., segmentation fault after the ROP execution). However, in case of software watermarking, the following issues have to be considered:

1. since the embedded ROP is for presenting a watermark, the payload has to be built all by the program itself;
2. code used in constructing the payload should not appear to be suspicious when compared to the original program;
3. after retrieving the watermark, ROP should return control to normal execution without sabotaging the program's normal execution.

Besides these, we also need to bypass protection mechanisms, e.g., Address Space Layout Randomization (ASLR). Clearly the old-fashioned way of launching an ROP for malicious attack is inappropriate in ROP-based watermarking. Instead, we need to propose a customized technique for our purposes. In the following subsections, we present each step of our scheme in more detail.

3.3 Locating watermarking gadgets

As mentioned in Section 3.2, ROP execution is constructed on the basis of instruction pieces called gadgets located from existing code regions, thus our first step is to find gadgets useful in watermark generation. However, using instructions inside the program itself is susceptible to simple program transformation attacks (as discussed in Section 2.3).

To provide resistance to the transformation attacks, we make use of the shared libraries (e.g. `libc.so` for UNIX and `kernel32.dll` for Windows). Such libraries are linked to almost any programs and cannot be easily modified. Therefore, we choose to search for watermarking gadgets from these system libraries, since semantic-preserving transformations on the watermarked program cannot prevent it from correctly targeting gadgets inside such legacy code.

Because we are using ROP only to hide watermarking behavior, gadgets to be executed needs to have no more than the following functions:

Register loading to load watermark messages and target memory addresses into registers. Note that this may not be as simple as a `<pop; ret>` sequence. Arithmetic/logical/shifting operations are also included.

Memory writing to write the value of source register to the memory area pointed to by the address in the target register, e.g., `<mov [eax], ecx; ret>`.

Stack shifting to control stack pointer, allowing watermarking ROP to be linked without being redirected with additional code. These are basically special register loading gadgets, since their only target is the stack pointer register.

Transferring to record the current position of instruction pointer when the program is directed to ROP, and to recover after the watermarking code is executed. Besides a specific combination of the former types of gadgets, we specifically exploit register exchange gadgets like `<xchg eax, esp; ret>` here since they can store the current stack pointer for later usage.

Since the introduction of ROP, many have proposed automatic searching of useful instruction sequences [15,26,29]. We make use of these existing techniques to locate the gadgets we need from system libraries. To be compatible with ASLR-enabled systems, we record gadgets by their offsets from entries of certain functions in the libraries, so that at runtime their absolute address can be computed easily with the assistant of function pointers.

[3]Note that it is only for simplicity that the watermark message is directly written in the ROP payload. In real implementation, this can be easily avoided with more complex gadget combinations.

Projects/libs	Size(KB)	Gadget types			
		Register loading	Memory writing	Stack shifting	Transferring
bzip2	63.488	22	2	2	1
hmmer	204.288	8	*NA*	*NA*	1
lbm	24.064	8	*NA*	*NA*	1
mcf	18.432	8	*NA*	*NA*	1
sjeng	105.984	12	1	1	1
soplex	306.688	24	*NA*	*NA*	1
libc.so.6	1335.56	9	5	2	4
kernel32.dll	857.6	13	2	8	3

Table 1: Number of available gadgets in projects and libraries

We search these four types of gadgets in a collection of SPECint-2006 benchmark projects as well as two system libraries `libc.so.6` and `kernel32.dll`. Only gadgets that complete a functional chain are recorded as available. Results shown in Table 1 confirm that both system libraries provide fully functional gadget sets for watermarking, while the same searching on small program modules are likely to fail. In addition, we found that typically gadgets of the equivalent function can be found at more than one locations, which could provide flexibility in formatting the payload.

3.4 Distributive and dynamic construction of watermarking payload

Constructing a reasonable watermark in ROP typically requires to execute dozens of gadgets, thus the payload used to chain the gadgets up would be of notable length. The code used to construct such payload would then be even longer, leaving potential targets for attackers. However, as briefly mentioned in Section 1, an important advantage of our ROP-based watermarking technique is to be able to spread out the watermarking components throughout the execution path to minimize suspicion from program analysis. Our design splits the watermarking payload into small segments to be constructed in different functions of the program which we called "carriers". With a number of carriers, we managed to embed just a small piece of code in each carrier that controls only a few gadgets, largely reducing the suspicion raised since the inserted code is almost negligible compared to the size of the original carriers. Refer to Section 4 for security evaluations on this.

We use TEMU, the dynamic analysis component of BitBlaze binary analysis platform [30], to trace the execution of the program with the secret input and to record all functions that are executed on the path along with their size, invoking frequency, and a call graph showing the control flow among them. Each function on this execution path is a carrier candidate for the watermarking payload. We sort the carriers by their size and embed longer segments of the payload in larger carriers since they provide better cover for the additional code.

Instead of storing the watermarking payload inside static data region, we construct the payload segments "on the fly". That is, only when a carrier is executed will the payload segment it carries be created in the program's heap area (we choose not to do this in carriers' stack frame in order to minimize the affect to program's normal execution caused by watermarking). We diversify the forms in which the payload segments are constructed to make it harder to recognize a pattern that indicates watermarking. Payload segments could be arranged as

- Integer arrays;
- Character strings;
- Selected variables of newly created instances of composite data structures (e.g., C structs and C++ classes).

Figure 3 shows examples on how the watermarking payload stored in a linear memory space can be built with different types of program objects. As we can see, the same piece of payload corresponding to the gadget `<pop eax; ret>` could stand in different ways: an integer array with two elements, two variables next to each other in an instance of a struct/class. We construct the payload based on the data structures that are already in the subject program and do not create new types of structures serving solely for watermarking which could be targets for attackers.

Furthermore, since the payload segments are formatted as heap-allocated data pieces, it's easy to use them in computing other variables owned by the program. Doing so creates active connections between the watermark components and the subject program, making them more difficult to be disabled even if they are somehow spotted by the adversary.

3.5 Payload chaining via stack pointer manipulation

Although splitting watermarking payload into short segments and distributing them among multiple carriers improve security, we still need a way to ensure that the discretely distributed payload pieces can be chained into a continuous stream during watermark extraction so that the ROP execution under their control works correctly.

As shown in Figure 4, when the watermarking payload is cut into segments, we attach stack-shifting gadgets at the end of them. In this way, each of the segments is responsible to relocate the stack frame correctly to the exact memory address of the next one, so that the watermarking payload works as if it were a continuous piece.

As discussed in the previous subsection, segments of the watermarking payload are constructed dynamically. During runtime whene one of them is constructed, the watermarked program checks if its previous and/or following segment had already been constructed. If either of them exists, the program chains the segments together by updating their stack shifting gadgets. Since the segments are generated one after

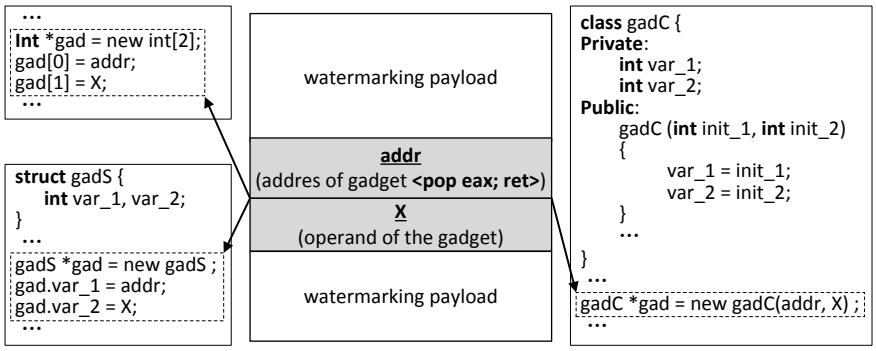


Figure 3: The diversity of payload formatting in ROP-based watermarking

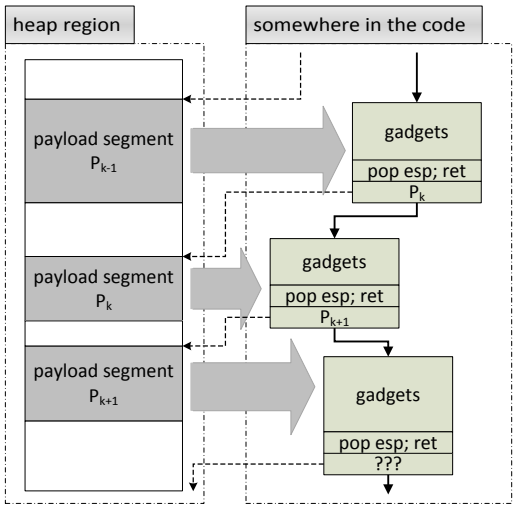


Figure 4: Chaining discrete payload segments via stack shifting gadgets (The dash arrows indicate stack pointer relocation while solid arrows indicate ROP execution flow)

another (although they may not be in the strict order as when they are used), each of them would either link itself to its neighbors or get linked by the neighbors.

3.6 Triggering ROP via function pointer overwriting

Now that the watermarking payload is chained up for execution. The last missing piece in the ROP-based watermarking is to instantiate the execution of the hidden watermarking path. This involves transitioning from normal execution of the program to ROP execution of the pre-selected gadgets, and we don't want instructions that perform such a transition to look suspicious.

We use a function pointer for triggering the ROP execution. The reason of this design choice is because a function pointer not only allows control transfer to the ROP gadgets via a simple overwriting of its value, but also provides a natural way of loading the initiating stub of the watermarking payload (given in the example in Section 3.2) directly onto the stack as function parameters. Usually this watermarking trigger points to a dummy function we added to the program so that in normal execution the trigger calls the dummy function and performs computations that do not affect the rest of the program. Upon inputting the secret input, the value of trigger is overwritten to point to the first gadget of the initiating stub (while the other parts of this stub will be pushed on the stack later by the trigger). Later when the trigger is invoked, gadgets of the initiating stub get executed, which save the current environment and transfer control to the first segment of watermarking payload.

The next question is how to encode the trigger condition (testing the secret input). A simple solution is to use a conditional block (e.g., an `if` statement) to compare current program input with the constant secret input. However, such conditional block introduces branching in the control flow graph and could attract attention of program analysis. Here we propose a novel idea to conceal such branching, see Figure 5. We exploit the right shifting operation so that only when $Input$ equals to Key will the variable x be set to 0. As a result, the program is able to conditionally determine the function pointer's value without explicitly introducing control flow transfer instructions.

```
void RopTrigger_A (int Input, int Key)
{
    int a=Input-Key;
    int b=Key-Input;
    a>>31;
    b>>31:
    /* if Input==Key, then x=0, otherwise x
       =-1 */
    int x=a^b;
    /* overwriting */
    &funcPt=addressA*(0-x)+addressB*(1+x);
    funcPt(Initial_Gadgets);
}
```

Figure 5: Testing trigger condition without branching

Note that the above example is only a raw implementation of our watermark trigger. In practice there are many ways to build the same function in more complex form so that the embedded trigger is harder to locate or analyze. For example, we can use the hash value of Key to compute a and b as similarly suggested in [28]. Simple right shifting can be replaced with equivalent arithmetic combinations. Pointer aliasing may also be applied to the function pointer overwriting. Finally, code block of the trigger can be taken apart and merged with another functional module to decrease its visibility.

3.7 Implementation

Our ROP-based watermarking toolkit is implemented in a set of python scripts consisting of a *gadget scanner*, a *trace processor* and a *code re-writer*, as shown in Figure 6.

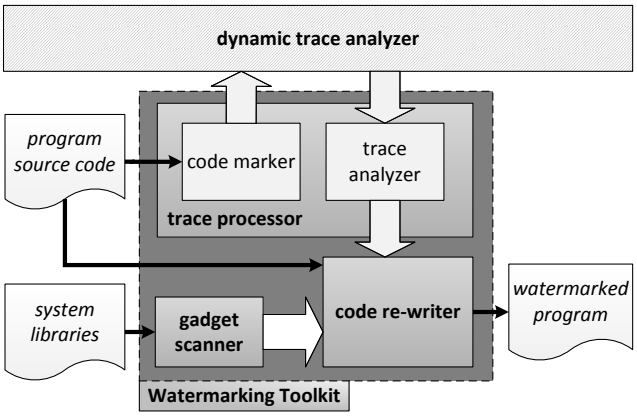


Figure 6: Implementation of ROP-based watermarking

Gadget scanner analyzes the system libraries, selects available gadgets of the four types given in Section 3.3 and creates a gadget database. It also generates the ROP payload that chains the candidate gadgets into the watermarking code.

Meanwhile, the trace processor analyzes the execution traces of the subject program under the secret input to find available carriers for watermarking payload. It not only finds the entry points of carriers but also records which parts of them are executed in the given trace.

After these preparations, code re-writer modifies the program's source code to embed the watermark. First, it divides the payload provided by the gadget scanner into segments and generates code for building and linking them. Following that, it distributes these code into the carriers, and also plants the dummy function as well as the ROP trigger. Finally, the modified source code is re-compiled, resulting the watermarked program.

Secret trace analysis is a tricky part of the watermarking since the analyzer only records execution on *binary* level while the watermark is embedded to the *source code*. To fill this gap, our trace processor consists of a *code marker* and a *trace analyzer*.

First, code marker makes a special copy of the program's source code by marking entry points and end points of functions as well as conditional code blocks in them with ineffective inline assembly sequences (which we called *tags*). We format these tags so that they are not only easy to be recognized, but also tell the exact locations in the corresponding source files they are assigned to. In particular, we let each tag to be started and ended with a `<mov edx, edx>` instruction, which never occurs in regular binaries. Additionally, a global junk variable is added in the program, with each tag carrying instructions that write the file ID and line number of the marked position in sequence to this junk variable. consequently, the resulting executable from the marked source is run with the secret trace recorded, and the trace analyzer can simply search for the inserted tags to figure out exactly which part of the source code is executed.

The trace processor is implemented as a total of 207 lines of scripts, in which the code marker takes 135 lines and the trace analyzer takes 72 lines. Code re-writer script consists of 191 lines due to the need of generating code for payload preparation in addition to scanning and re-writing source files. Gadget scanner is extended from GALILEO [27], an existing gadget searching algorithm.

4. EVALUATION

In this section, we subject our ROP-based watermarking to a number of security analyses and present the results. We also measure the static and dynamic overhead.

Security of software watermarking is usually evaluated in terms of the *stealth*, *credibility*, and *resilience*. Stealth typically refers to how well the watermark blends in the code or data around it; credibility describes how precisely the watermark can be retrieved; while resilience measures the resistance against determined attempts at discovery or removal. ROP-based watermarking extracts a watermark as a direct result of executing the embedded ROP gadgets; thus its credibility is quite self-evident. Therefore, in this section, we mainly focus on analyzing stealth and resilience of our approach. Meanwhile, the overhead caused by applying the ROP-based watermarking is also evaluated on three aspects – time it takes to generate the watermark, increment in code size, and the additional heap space required.

We apply our scheme on a number of subject programs from the SPECint-2006 test suite which are subject programs that previous research chose to work with [23]. The experimental watermarking is designed to simply output a watermark message on the screen. All tests were run on a PC with a 2.66GHz Intel Core 2 Quad CPU, 4GB memory, and Windows 7 operating system.

4.1 Static stealth

Static stealth of software watermarking measures how well a watermark fits into the program around it. As mentioned in Section 3, the ROP-based watermarking needs to insert a small amount of code into the program for constructing its payload. Therefore, static stealth plays an important role in evaluating our design. The good news is that code introduced by our ROP-based watermarking is only for creating and updating the payload segments which appear to be program data. It is broken down to instruction level and distributed over many carriers in the program (see Section 3.4). Considering the diversity of such code as also discussed in Section 3.4, we expect that ROP-based watermarking presents good static stealth in this measurement.

We adopt the static stealth measurement introduced by Collberg et al. [11], briefly described as follows:

- First, given a piece of watermarking code that generates an x-bit watermark, a *dictionary* is created as an instruction level profile. This is done by sweeping a peephole (of which the size is set to one to three instructions as suggested by Collberg et al.) over the instruction stream of the watermarking code, while distinct instruction combinations observed in the scanning are added as new *words* to the dictionary.

- Second, to measure whether the watermarking code is stealthy in a program, all words in the dictionary are then searched in the original and watermarked programs. The more words appearing in them, the better the watermarking code fits into the program.

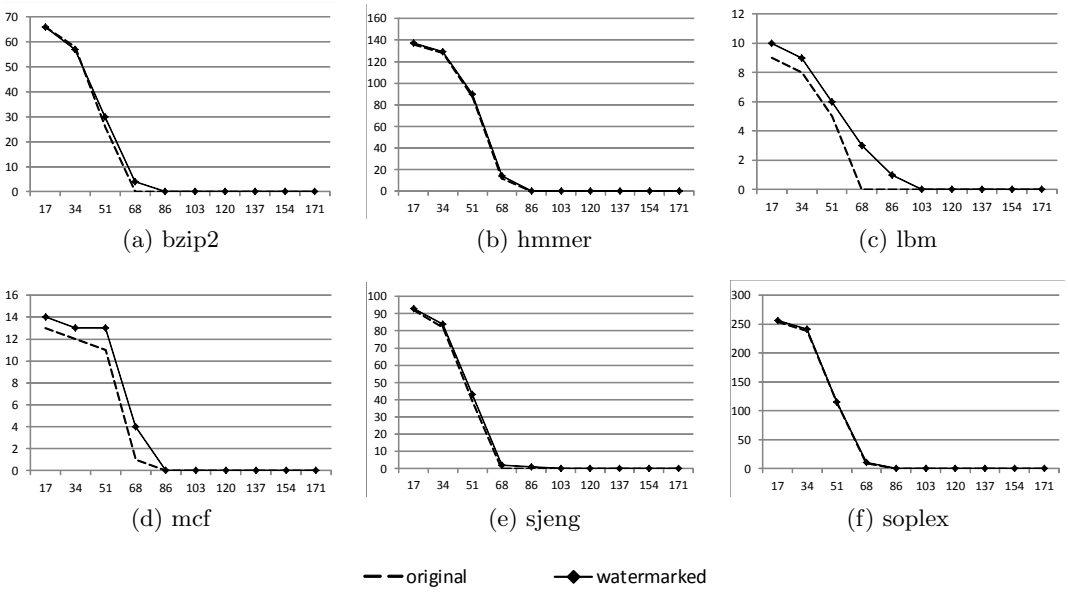


Figure 8: Locality evaluation on static stealth

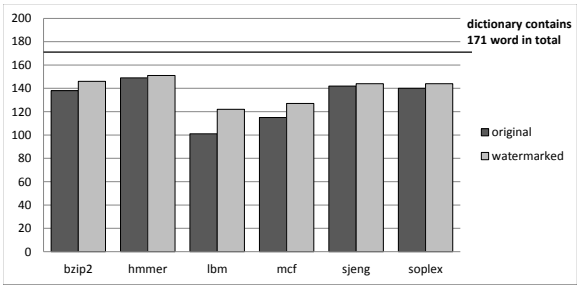


Figure 7: Static stealth of ROP-based watermarking

We apply our ROP-based watermarking on six benchmark programs `bzip2`, `hmmer`, `lbm`, `mcf`, `sjeng`, and `soplex` to embed a 192-bit watermark. By analyzing the watermarking code inserted into the six watermarked program, we identify a total of 171 words in the dictionary. Figure 7 shows the number of words in the dictionary that are found in the original and watermarked programs.

There are two important takeaways from these results. First, we observe that between 70 and 85 percent of the words in the dictionary of watermarking code already exists in the original programs. This shows that most of the code introduced by our ROP-based watermarking can be seen in the ordinary program, making it hard for attackers to look for special instruction sequences that could serve as signatures for detection. Second, we find that even in the watermarked programs, there are only 72 to 88 percent of the words in the dictionary being observed. Meanwhile, the differences between results on the original programs and the watermarked programs are pretty small – making the original and watermarked programs relatively indistinguishable in this analysis.

We further extend this experiment to perform a locality evaluation by running a considerably large sliding window (200 instructions in our experiment, since the watermarking code itself only has 271 instructions in total) over the programs' instruction stream. Again, we look for words from the dictionary when running the sliding window, one step to the right when the number of words found is below a threshold, or 200 steps to the right otherwise to avoid double counting. Locations where the number of words found exceeds the threshold are considered "hotspots" that have similar profile compared to the watermarking code, and will be marked as suspicious. Figure 8 shows the result where the horizontal axis shows different threshold settings and the vertical axis shows the number of "hotspots" found.

Results are consistent with those from the previous experiment, where few windows in which over 50% coverage of watermarking dictionary are reported. When comparing the results for the original and the watermarked programs, there is no significant increase on the number of hotspots.

4.2 Dependence analysis on watermarking components

In the second analysis on the stealth of the watermarking technique, we assume a more realistic scenario where the attacker manages to locate one distinguishable component used in the watermark, and tries to trace the other parts of it with dependency analysis to the exposed component. Existing watermarking schemes can be vulnerable on this aspect since certain features used by their external extractor to recognize watermark may also be exploited by adversaries, while the connections between their code and data are directly exposed to analyses.

As an example, we simulate such a dependency analysis on the CT watermarking technique [10], one representative watermarking techniques proposed in the literature. In the simulation, we set the root node of the heap-allocated watermark graph to be static, as if it is recognized already. IDA pro 6.4 is then used to launch a dependency analysis starting from the given root node in order to find anything

that are semantically connected. Figure 9 shows the analysis result, where we find that based on the exposed root node, IDA pro is able to detect a dependency graph connecting all functions that participate in the watermark generation, making it relatively easy for attackers to narrow down to specific functions in searching for the watermarking code.

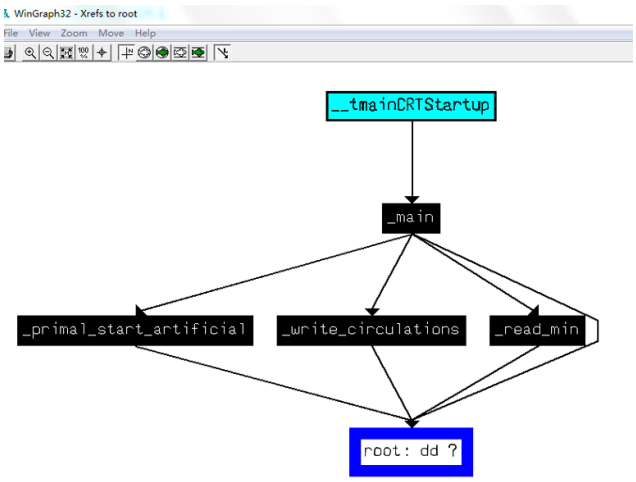


Figure 9: Dependency analysis result of CT watermarking

In contrast, our ROP-based watermarking payload stay in the data region of the program, thus from normal perspective it has no explicit connection to the corresponding gadgets, making the watermarking semantics "invisible" to code analyses (including dependence analysis), i.e. the code stream in charge of writing the watermark message into program's memory then printing it out literally does not exist when the embedded ROP is not triggered. For verification, we perform a similar simulation on our design, in which we intentionally let the watermarking code write the hidden message in a *static* character string that can be easily picked up by a static analyzer, and again look for semantically connected portions from the program. However, IDA pro encounters a failure to reach anything that have dependencies on the exposed string, i.e., it believes that semantically there is no instruction in the watermarked program that tries to read or modify the string at all.

To confirm this result, we dynamically execute the watermarked program and monitor the target string to find out instructions that operate upon it. Figure 10 shows screenshots of this dynamic analysis (before and after the exposed character string is overwritten). We can see that instructions operating on the target string are actually unintended instructions from the shared library that are beyond the scope of the analyzer. This confirms that there is zero dependency found in the watermarked program.

Note that the above simulation is only for demonstrating the difference on semantic visibility between our design and the previous work. In our ROP-based watermarking, we do not actually use such a static string to store the watermark message, and therefore the dependency analysis might even have no ground to begin with.

4.3 Resilience

In this subsection, we discuss the strength of our design against distortion attacks in which the adversary attempts to destroy the watermark by twisting the binary of watermarked program. Common binary obfuscations (including function and basic block reordering, function inlining and outlining, data restructuring, etc.) are the main approaches of distortion attack while program packing and optimizing are also included in the adversaries' arsenal.

Intuitively, semantic-preserving transformations do not affect our design because they do not change the way gadgets are constructed. One possibility, though, is to target parameters of the function pointer for triggering ROP execution. However, the trigger function pointer works via a indirect call, whose target is hard to be determined at binary level. Theoretically, a complete distortion on the trigger needs to locate all indirect calls and modify all their potential targets including those imported to the program (which are highly error-prone). This makes the adversary's task impractical.

We test the resilience of all our watermarked programs against transformation of a selection of well-known tools:

- *xenocode*, performs binary-level obfuscation including encrypting static strings, randomly inserting redundant ineffective instructions, and obfuscating program's control flow [3];
- *UPX*, provides high-quality packing and compression on softwares [2]; and
- *LLVM optimizer*, enables various source- and target-independent binary optimizations, some of which perform decompilation/recompilation on target code [1].

Results show that after transformations, the watermarked programs can still correctly generate and chain the watermarking payload, and the hidden watermarks can still be correctly extracted. This not only indicates the good resilience of our design, but also shows that our ROP-based watermarking has potentially high compatibility should we want to combine it with other protection mechanisms.

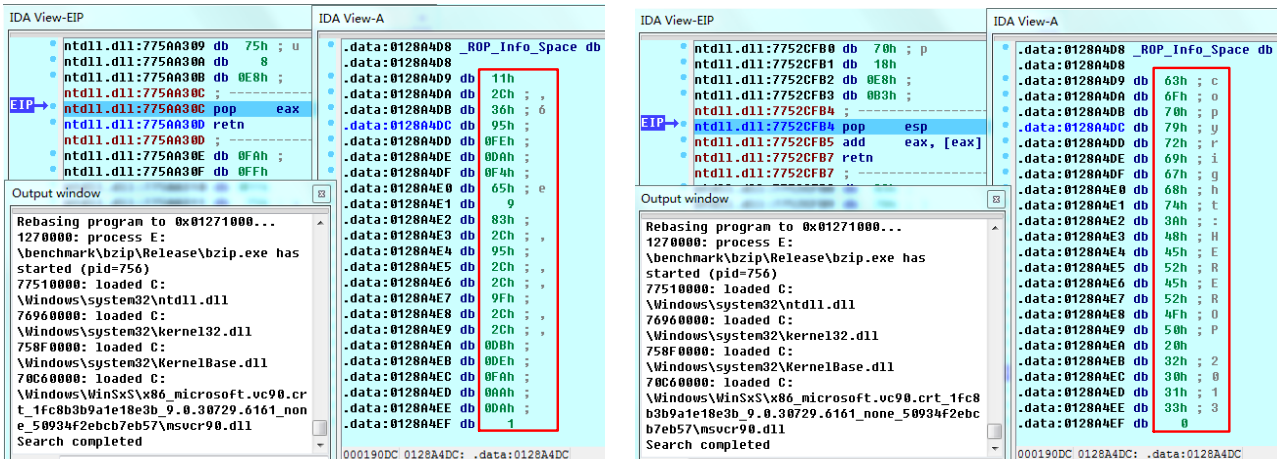
4.4 Overhead

In this subsection, we evaluate the performance overhead of our design in comparison with graph-based watermarking. Intuitively, ROP-based watermarking involves far less overhead in both execution time and heap space because it does not need to encode the watermark into complex data structures (e.g., a graph).

In our experiments, we apply the CT watermarking technique to encode the watermark into radix graphs because it is widely considered to be the most efficient solution in graph-based watermarking [11]. All programs involved in the test are compiled under the same setting, and programs watermarked with both methods do not have any additional protection mechanisms integrated. Table 2 shows the result for runtime overhead, while Figure 11 presents results for static program size increments.

We find that the runtime overhead of a ROP-based watermarked programs is significantly smaller than that in the graph-based one – in some cases more than two orders of magnitude smaller. Our method also experiences smaller increase in program size.

We also evaluate the size of heap-allocated data structures that are constructed for the watermark. Results are very much identical for all six programs, in which CT watermarking uses 1,536 bytes of heap space to encode a 192-bit



(a) Before string overwriting

(b) After string overwriting

Figure 10: Unintended instructions that overwrite the watermark string in our design

benchmark	runtime overhead (ms)	
	ROP-based watermarking	graph-based watermarking
bzip2	1.33	154.94
hmmer	6.48	10897.63
lbm	536.61	41363.07
mcf	88.62	47177.45
sjeng	19.04	4262.94
soplex	2.03	4.61

Table 2: Runtime overhead

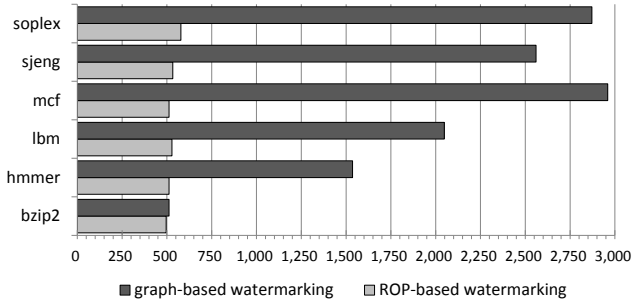


Figure 11: Increment in program size (bytes)

watermark into a radix graph, whereas our ROP-based watermarking uses 156 or 188 bytes when the gadgets are formatted in integer arrays or in structs/classes, respectively. Again, our ROP-based watermarking has a clear advantage.

4.5 Our method vs. RopSteg

In this subsection, we give a comparison between our method and RopSteg when applied to hide software watermark. As discussed in Section 2, RopSteg is a general tool for hiding code portions of a program with ROP. It has a different threat model compared to software watermarking and is susceptible to simple program transformation attacks. In this subsection, we, instead, focus on two evaluations. First, the amount of additional instructions inserted that participate dynamically in normal runs of the program (without watermark generation) and in watermark generation. This is an important security evaluation of software watermarking because instructions dedicated to watermark generation (those not participated in normal runs of the program at all) are suspicious and attract program analysis. Second, we also evaluate the program size increments. Table 3 shows the average result for the six benchmark programs we have tested, since the results are very much constant among them (shown in the previous subsection for our method and discussed in [17] for RopSteg).

Our ROP-based watermarking is designed based on an idea that instructions introduced by watermarking should be amphibious – they should be executed in regular runs of the program (without watermark generation). Our evaluation (second column of Table 3 confirms that our design meets this criteria, as 100% of the newly added instructions participate in normal runs of the programs. Meanwhile, only 61% of the instructions inserted by RopSteg participated in normal executions of the programs. In addition, the total size increment caused by RopSteg is larger than our method.

5. DISCUSSION

5.1 ROP defenses

A number of ROP defenses have been proposed to detect and stop ROP execution [6,7,13,22,24,33]. One can imagine that if triggered on a system with ROP defense deployed, the embedded ROP execution in the program transformed with our ROP-based watermarking would set off an alarm, and that specific execution where watermark extraction took place might be terminated.

However, watermark extraction is a special scenario that only happens when certain concerned party tries to either prove the software's ownership or to identify the authorized users. That's why a special input is used to trigger the watermark execution. In our ROP-based watermarking, nor-

Method	Newly inserted instructions executed		Program size increase
	Normal run (without watermark generation)	Watermark generation	
Our work	100%	0%	512
RopSteg	61.5%	38.5%	650

Table 3: Overhead comparison between our method and RopSteg

mal executions of the watermarked program will show no characteristics of ROP execution at all, and therefore do not conflict with ROP defenses. Although it is true that extracting watermark hidden with our design presents ROP behaviors, we believe that given that watermark extraction is such a special event, it is reasonable to simply run the watermarked program in a specialized environment or temporarily turn off the ROP defense.

5.2 Compatibility

Our design is implemented and evaluated under the x86 instruction architecture. Effectiveness and efficiency on other architectures, such as SPARC or ARM, might vary from our current results. That said, there have been reports of successful ROP on various platforms [4, 14, 16], which suggest that our proposal could work on these platforms, too.

Nevertheless, our design makes use of gadgets from shared libraries, suggesting that the watermark extraction depends on the execution environment, e.g. library versions, to correctly re-build the ROP execution path. One possible solution is to source for multiple sets of watermarking payload strings, each corresponding to a distinct version of the selected libraries, so that the watermarked program can detect the environment on which it is running and point the watermarking trigger to the corresponding watermarking payload. We leave this as future work.

5.3 Library Replacement Attack

At this moment, a potential weakness of our ROP-based watermarking is the so-called library replacement attack, i.e. the adversary replaces the original dynamic libraries that would link to the watermarked program, say L, with its custom library L' that may be bundled in the software package. So long as there are gadgets locate in L, such replacement could render errors during ROP and therefore compromise the watermark recovery process. In the extreme case, the adversary could replace all dynamic libraries just in case it does not know where the gadgets reside. We plan to improve our approach on this aspect in the near future. Possible ways might include exploiting gadgets inside libraries that cannot be circumvented (this is possible in some operation system, e.g. any program runs on windows system must load `Kernel32.dll` which provides system API), or introducing tamper-proofing approaches that at runtime check whether the loaded libraries are compromised.

6. CONCLUSION

We proposed a novel dynamic software watermarking design that embeds and exhibits watermark through a memory error exploiting technique named return-oriented programming. Our ROP-based watermarking is able to transform important watermarking code into ROP gadgets and build them in the data region. The watermark can be extracted by activating ROP execution along these gadgets constructed. Evaluations show that compared to previous works, our design achieves better stealth because of its fine-grained code distribution. Analysis and experiments also suggest that our design not only presents good resilience against attacks with code obfuscation and re-packing, but also causes notably lower overhead.

We take our ROP-based watermarking as a successful attempt of turning a malicious approach (return-oriented programming in this case) into a benign usage. We also believe that this work opens a rather different view for software watermarking – instead of hiding messages in special executions, we could make execution itself invisible.

7. ACKNOWLEDGMENTS

This project is partly supported by the National Key Basic Research Program of China (Grant No. 2013CB834204), the National Natural Science Foundation of China (Grant No. 61272423, 61303213), the Natural Science Foundation of Tianjin (Grant No. 14JCYBJC15300), and the Open Project Foundation of Information Security Evaluation Center of Civil Aviation, Civil Aviation University of China (Grant No. CAAC-ISECCA-201403).

8. REFERENCES

[1] The llvm compiler infrastructure. http://llvm.org/.

[2] Upx: the ultimate packer for executables. http://upx.sourceforge.net/.

[3] Xenocode. http://www.xenocode.com.

[4] E. Buchanan, H. Roemer, H. Shacham, and S. Savage. When good instructions go bad: Generalizing return-oriented programming to risc. In *Proceedings of the 15th ACM conference on Computer and communications security (CCS)*, pages 27–38, 2008.

[5] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy. Return-oriented programming without returns. In *Proceedings of the 17th ACM conference on Computer and communications security (CCS)*, pages 559–572, 2010.

[6] P. Chen, H. Xiao, X. Shen, X. Yin, B. Mao, and L. Xie. Drop: Detecting return-oriented programming malicious code. In *Proceedings of the 5th International Conference of Information Systems Security (ICISS)*, pages 163–177, 2009.

[7] Y. Cheng, Z. Zhou, M. Yu, X. Ding, and R. H. Deng. Ropecker: A generic and practical approach for defending against rop attacks. In *Symposium on Network and Distributed System Security (NDSS)*, 2014.

[8] C. Collberg, E. Carter, S. Debray, A. Huntwork, J. Kececioglu, C. Linn, and M. Stepp. Dynamic path-based software watermarking. In *Proceedings of*

the ACM SIGPLAN 2004 conference on Programming language design and implementation (PLDI), pages 107–118, 2004.

[9] C. Collberg and J. Nagra. *Surreptitious Software — Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Software Security Series. Addison-Wesley, 2009.

[10] 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)*, pages 311–324, 1999.

[11] C. Collberg, C. Thomborson, and G. M. Townsend. Dynamic graph-based software watermarking. Technical Report TR04-08, Department of Computer Science, The University of Arizona, 2004.

[12] P. Cousot and R. Cousot. An abstract interpretation-based framework for software watermarking. In *Proceedings of the 31th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 173–185, 2004.

[13] L. Davi, A. Sadeghiy, and M. Winandyz. Ropdefender: A detection tool to defend against return-oriented programming attacks. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, pages 40–51, 2011.

[14] A. Francillon and C. Castelluccia. Code injection attacks on harvard-architecture devices. In *Proceedings of the 15th ACM conference on Computer and communications security (CCS)*, pages 15–26, 2008.

[15] T. Holz and F. Freiling. Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms. In *Proceedings of the 19nd USENIX conference on Security (USENIX Security)*, pages 383–398, 2009.

[16] T. Kornau. Return oriented programming for the arm architecture. Master's thesis, Ruhr-Universität, Bochum, 2010.

[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)*, pages 265–272, 2014.

[18] G. Myles and C. Collberg. Software watermarking through register allocation: Implementation, analysis, and attacks. In *Proceedings of the 6th International Conference of Information Security and Cryptology (ICISC)*, pages 274–293, 2003.

[19] G. Myles and C. Collberg. Software watermarking via opaque predicates: Implementation, analysis, and attacks. *Electronic Commerce Research*, 6(2):155–171, 2006.

[20] G. Myles and H. Jin. Self-validating branch-based software watermarking. In *Proceedings of the 7th International Workshop of Information Hiding (IH)*, pages 342–356, 2005.

[21] J. Nagra and C. Thomborson. Threading software watermarks. In *Proceedings of the 6th International Workshop of Information Hiding (IH)*, pages 208–223, 2004.

[22] K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarotti, and E. Kirda. G-free: defeating return-oriented programming through gadget-less binaries. In *Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC)*, pages 49–58, 2010.

[23] J. Palsberg, S. Krishnaswamy, M. Kwon, D. Ma, Q. Shao, and Y. Zhang. Experience with software watermarking. In *Proceedings of the 16th Annual Conference of Computer Security Applications (ACSAC)*, pages 308–316, 2000.

[24] V. Pappas. kbouncer: Efficient and transparent rop mitigation. Technical report, Columbia University, 2012.

[25] C. Ren, K. Chen, and P. Liu. Droidmarking: Resilient software watermarking for impeding android application repackaging. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering (ASE)*, pages 635–646, 2014.

[26] R. Roemer, E. Buchanan, H. Shacham, and S. Savage. Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 15(1):2, 2012.

[27] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and communications security (CCS)*, pages 552–561, 2007.

[28] M. Sharif, A. Lanzi, J. Giffin, and W. Lee. Impeding malware analysis using conditional code obfuscation. In *Proceedings of the 16th Annual Network & Distributed System Security Symposium (NDSS)*, 2008.

[29] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and C. Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *Proceedings of the 34rd IEEE Symposium on Security and Privacy (S&P)*, pages 574–588, 2013.

[30] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. Bitblaze: A new approach to computer security via binary analysis. In *Proceedings of the 4th International Conference on Information Systems Security (ICISS)*, pages 1–25, 2008.

[31] R. Venkatesan, V. Vazirani, and S.Sinha. A graph theoretic approach to software watermarking. In *Proceedings of the 4th International Workshop of Information Hiding (IH)*, pages 157–168, 2001.

[32] T. Wang, K. Lu, L. Lu, S. Chung, and W. Lee. Jekyll on ios: When benign apps become evil. In *Proceedings of the 22nd USENIX conference on Security (USENIX Security)*, pages 559–572, 2013.

[33] C. Zhang, T. Wei, Z. Chen, L. Duan, S. McCamant, L. Szekeres, D. Song, and W. Zou. Practical control flow integrity & randomization for binary executables. In *Proceedings of the 34rd IEEE Symposium on Security and Privacy (S&P)*, pages 559–573, 2013.

[34] W. Zhou, X. Zhang, and X. Jiang. Appink: Watermarking android apps for repackaging deterrence. In *Proceedings of the 8th ACM Symposium on InformAtion, Computer and Communications Security (ASIACCS)*, pages 1–12, 2013.