NATIONAL UNIVERSITY OF SINGAPORE

SCHOOL OF COMPUTING

DOTA - 2022

Defense of the Ancients

The Projects for DOTA2022
(Defense of the Ancients)
Singapore, July 2022.

Table of Contents

Demonstration of Meltdown and Spectre

Dazhi Feng
Southern University of Science and Technology
12012726@mail.sustech.edu.cn

Shenghao Xie
Wuhan University
XieShenghao020508@163.com

Yangfan Liu
University of Electronic Science and Technology of China
lyf910910@163.com

Junmeng Liu
Xi’an Jiaotong-liverpool University
Junmeng.LIU20@student.xjtlu.edu.cn

Xuyue Liu
Huazhong Agricultural University
xy142899@163.com

ABSTRACT

Modern computer systems are susceptible to Meltdown and Spectre attacks, as malicious programs can use both attacks to gain access to confidential information stored in the memory of other running programs. In this paper, we explain the principle and attack process of the two attacks based on the two characteristics of CPU: Out-of-Order Execution and Speculative Execution. Subsequently, we reproduce the attack in Ubuntu 16.04 of the personal computer, and it successfully steals the secret value hidden in the kernel address, thus it verifies the effectiveness of the vulnerability. Finally, we introduce KPTI and PASD schemes to separately defend against two kinds of attacks.

Categories and Subject Descriptors

B.4.1 [**Input/Output and Data Communications**]: Data Communications Devices; K.6.5 [**Management of Computing and Information Systems**]: Security and Protection

General Terms

Vulnerability, Security, CPU

Keywords

Meltdown, Spectre, Out-of-order Execution, Speculative Execution, Side Channel Attack (SCA)

1. INTRODUCTION

In January 2018, the Meltdown and Spectre security incidents were a big deal across the security community. The Meltdown vulnerability affected almost all Intel CPUs and some ARM CPUs. The Spectre vulnerability affected all Intel CPUs and AMD CPUs, as well as mainstream ARM CPUs. From PCs, servers, and cloud servers to mobile smartphones, all are affected by these two sets of hardware vulnerabilities. It is these two high-impact CPU vulnerabilities that we are looking at.

The core idea of the Meltdown and Spectre attacks is to exploit the predictive and chaotic execution prevalent in modern CPUs. Predictive execution increases speed by executing multiple instructions at the same time. The order Meltdown and Spectre are both essentially cache-side channel-based attacks.

A side-channel attack exploits various physical state information closely related to internal operations that are leaked during the execution of a cryptographic algorithm. The attack is then combined with statistical techniques, for example.

Meltdown is a new attack that overcomes the problem of memory isolation altogether. It provides an easy way for any user process to read the entire kernel memory of the machine it is executing on, including all physical memory.

Spectre vulnerability is a vulnerability that can force other programs on a user’s operating system to access any location in their program’s computer memory space. The Spectre vulnerabilities are not a single easy-to-fix vulnerability but rather the sum of a class of potential vulnerabilities. They all exploit a side effect of ”predictive execution,” a common method used by modern microprocessors to reduce memory latency and speed up execution. Specifically, the Spectre vulnerability focuses on branch prediction, part of predictive execution.

We have conducted experiments and research on these two sets of high-impact hardware vulnerabilities and have successfully replicated both attacks.

2. FUNDAMENTS OF MELTDOWN/ SPECTRE

2.1 Out-of-order Execution

Rather than executing instructions in strict serial order, the CPU groups instructions in parallel based on correlation and finally aggregate the results of each group of instructions executed. The processor will execute the machine instructions in the order in which they are written in the program.

Execution in the order in which they are written is called sequential execution. When executing in writing order, a long-delayed instruction such as a load instruction that reads data from memory or a division instruction followed by an instruction that uses that instruction can result in a long wait. Although what can do nothing about this situation, sometimes the next instruction does not depend on the preceding instruction with a long delay and can be executed as soon as the operand is available. In this case, the order of the machine instructions can be disturbed so that even if the instruction is located later, it is executed first if it is executable, called jumbled execution. With out-of-order execution, instructions that cannot execute immediately due to data dependency are delayed, thus mitigating the effects of data disasters.

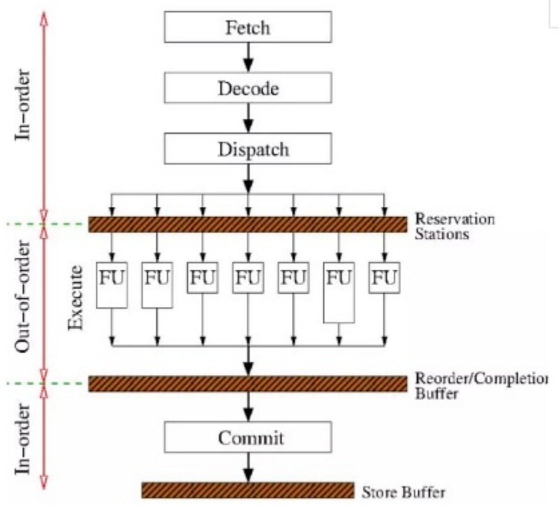


Figure 1: Out-of-order Execution

2.2 Speculative Execution

Speculative execution is when the CPU predicts the outcome of a conditional judgment based on the available information and then selects the corresponding branch for early execution. In the case of exceptions or incorrect branch predictions, the CPU discards the results of the previous execution, restores the CPU state to the correct state before the chaotic or speculative execution, and then selects the correct instruction to continue execution. This exception handling mechanism ensures that the program is executed correctly.[1] Still, the problem is that the CPU does not restore the contents of the cache when it restores the state, and these two sets of vulnerabilities exploit this design flaw to perform a channeling attack.

2.3 Side Channel Attack

Side channel attack – is a mode of attack that exploits information signals (e.g., power consumption, electromagnetic radiation, the sound of computer hardware running) inadvertently emitted by the computer to decipher. [2] For example, a hacker can read your display and file information on a disk through the electromagnetic radiation generated by the computer's display or hard drive. Or they were monitoring your computer by the different amounts of power required by computer components to execute specific programs; or knowing your account numbers and passwords just by the sound of keyboard taps. Side channel attacks, i.e., power consumption, EM (electromagnetic) radiation, and timing attacks are not actively involved but leak information through these interfaces. An adversary can easily detect secret information by performing simple and advanced analysis. Bypass attacks In SCA (bypass attacks), the attacker monitors/measures the physical characteristics of the chip (power, timing, current, etc.). During its regular operation. After collecting the required information, the attacker analyzes the collected data to determine secret information. Techniques such as Fourier analysis, hamming distance, simple power analysis, differential power analysis, and fault-sensitive attacks are commonly used.

2.4 Meltdown

2.4.1 Attack setting

In our attack, we consider the virtual machine based on personal computers. On the compromised system, the attacker has unrestricted code execution, which allows them to execute any code with regular user privileges. However, the attacker is physically unable to access the computer. We further assume that the system is fully secured by innovative software-based defenses, including ASLR and KASLR, and CPU features, including SMAP, SMEP, NX, and PXN. The most crucial assumption is that the operating system is bug-free.[1] Thus, no software vulnerabilities may be used to access the kernel or leak data. The attacker uses confidential user data, such as passwords, private keys, or other essential data.

Meltdown combines two fundamental components. First, the attacker forces the CPU to carry out a brief set of instructions using an elusive secret value hidden somewhere in physical memory. The brief instruction sequence serves as a hidden channel's transmitter, ultimately revealing the secret value to the attacker. Meltdown includes the following 3 steps:

Step 1: Catch the secret. Meltdown takes advantage of the fact that contemporary CPUs still carry instructions out of order during the brief interval between erroneous memory access and the triggering of an exception. Therefore, a register is filled with the content of a memory location selected by the attacker, which the attacker cannot access.

Step 2: Disseminating the secret. The out-of-order instruction sequence must be chosen to become a transitory instruction sequence. The transient instruction sequence includes indirect memory access to an address calculated using the secret (inaccessible) value to transmit the secret. Therefore, a transitory instruction uses the secret content of the register to access a cache line.

Step 3: Obtaining the secret. The attacker employs Flush +Reload to determine the accessed cache line and, as a result, the secret stored at the selected memory location. The corresponding memory of the probe array will be cleared out of the CPU cache through the clflush instruction initially. Subsequently, the exact one cache line of the probe array is cached when the transient instruction sequence of step 2 is run. The secret read in step 1 is the only factor determining where the cached cache line is located within the probe array. As a result, the attacker cycles through all pages of the probe array and calculates the access time for each page's first cache line (i.e., offset). The short access time indicates that the CPU has loaded the corresponding memory page into the cache while executing the above code. The secret value is related to the page number of the cached cache line.

An attacker can dump the entire memory by iterating over all addresses by repeating all three Meltdown steps.

2.5 Spectre

Similar to Meltdown, Spectre is based on the principle that when the CPU finds a branch prediction error, it discards the result of the branch execution and restores the CPU state, not the CPU Cache state. Specter can use this to break through inter-process access restrictions (e.g., browser sandboxing) to obtain data from other processes. Attacks such as Spectre are designed to break the isolation between different applications by exploiting an optimization technique known as speculative execution in CPU hardware implementations to trick programs into accessing arbitrary locations in memory, thereby leaking data in memory.

3. EXPERIMENT

In this section, we show how we simulated Meltdown and Spectre attacks, respectively. Then we discuss the difficulties during the demonstration.

3.1 Sample Meltdown Attack

The environment of the sample attack:

CPU	Intel Core i5-3240
OS	Ubuntu 16.04 ($GNU/Linux$4.8.0)

To begin with, we measure and compare the time to read the data from the cache and the memory. The aim is to prove that accessing data from the CPU cache is much faster than reading and writing data from memory. We first flush the CPU cache, read and write an element in an array to put it into the cache, and measure the CPU cycles used to access each element. [4]

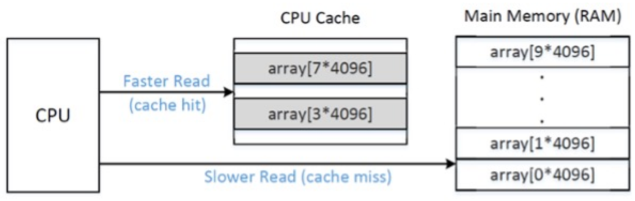
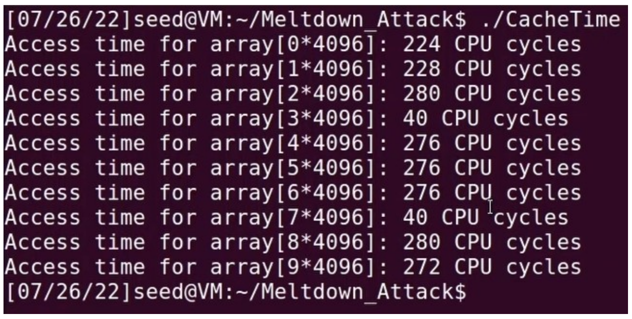


Figure 2: Cache and Memory[3]

In this case, we declare an array with 10 elements and access the elements with indices of 3 and 7 before we access all elements and evaluate the CPU cycles.

As for the experimenting environment, accessing the cache requires no more than 80 cycles, while accessing the memory requires significantly more cycles than 80. Therefore, if the CPU completes accessing a piece of data within 80 cycles on the experimenting machine, we assume the data is in the cache. After that, it is time to prepare for the Meltdown attack.



```
[07/26/22]seed@VM:~/Meltdown_Attack$ ./CacheTime
Access time for array[0*4096]: 224 CPU cycles
Access time for array[1*4096]: 228 CPU cycles
Access time for array[2*4096]: 280 CPU cycles
Access time for array[3*4096]: 40 CPU cycles
Access time for array[4*4096]: 276 CPU cycles
Access time for array[5*4096]: 276 CPU cycles
Access time for array[6*4096]: 276 CPU cycles
Access time for array[7*4096]: 40 CPU cycles
Access time for array[8*4096]: 280 CPU cycles
Access time for array[9*4096]: 272 CPU cycles
[07/26/22]seed@VM:~/Meltdown_Attack$
```

Figure 3: Cache Speed

The attack is based on two assumptions:

· We (the attacker) know the address of the secret.

· This secret cannot be obtained from the outside.

To demonstrate the attack, we put the secret data (the character "S," 94 in ASCII) in the kernel space, which is not accessible by user mode programs, using a kernel module. The module also saves the address of secret data in the kernel information buffer so that we can get the address using the 'dmesg' command. In a real-world attack, attackers find or guess the address by themselves. Moreover, to make the CPU accesses the secret faster and thus increase the possibility for the CPU to execute more instructions before it finds out that the secret cannot be leaked to user mode, we need to access the secret once to put it into the cache. Therefore, we create a file (/proc/secret_data) and a module function module_proc(), which provides an interface for user mode programs to interact with kernel modules. When a user mode program accesses the file, the kernel module function module_proc() will be called, which loads the secret variable to store the secret into the cache but will not return the secret value.[3]

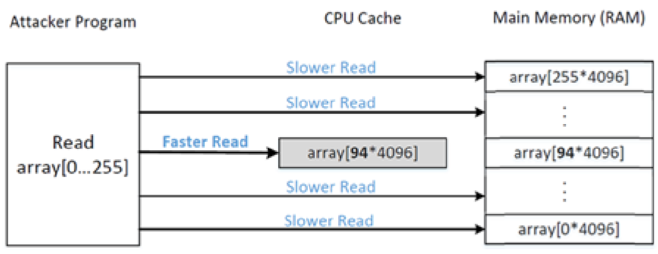
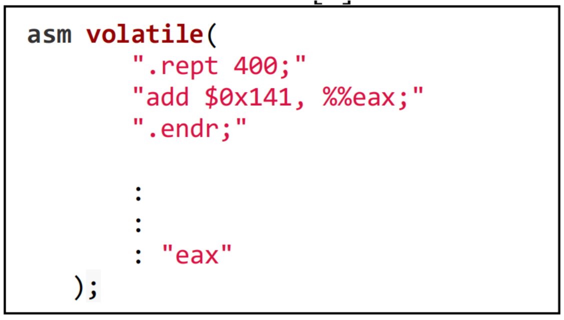


Figure 4: Reloading[3]

Then, we can launch the attack. We exploit the cache as a side channel through flushing and reloading. First, flush the cache so that the cache is empty. Then, try to access the secret and use the value in the secret. Here we use the secret value as an index to access the array. Since the secret address is inaccessible, the program will receive a SIGSEGV signal, which should be handled to prevent the program from being terminated. After this step, one element in the array is likely cached. Finally, the entire array should be reloaded, and the reload time of each element should be measured. If an element is loaded within 80 cycles, it has been in the cache. This element is likely to be the one accessed using the value of the secret. Therefore, we can infer that the index of this element is the secret.

The essential condition for the attack's success is that the CPU checks permission slowly enough to execute more out-of-order instructions. To retard the permission checking process and win the race condition, we can add a piece of assembly code before we access the secret[1]:



```
asm volatile(
        ".rept 400;"
        "add $0x141, %%eax;"
        ".endr;"

        :
        :
        : "eax"
   );
```

Moreover, to increase the accuracy of the attack, we can measure the access time repeatedly and take the index that the element has the shortest average access time as the secret.

3.2 Sample Spectre Attack

The experimenting environment:

CPU	Intel Core i7-10750H
OS	Ubuntu 20.04($GNU/Linux$5.4.0)

The first step of this attack, which is to count the CPU cycles used to access data in cache or memory, is the same as the Meltdown attack and will not be repeated here. It costs the experimenting machine no more than 200 cycles to access data in the cache and no less than 200 cycles to access data in memory.

Then we can launch the attack. Since obtaining data from other processes is much more complex, we demonstrate acquiring data from the same process that uses a getter to check whether the access is legal. The demonstration assumes that the process uses a getter function (the victim) to check permissions. To exploit branch prediction, the CPU is first trained to predict that the permission checking is passed by calling the victim with a valid argument multiple times.[4]Then flush the cache for later use. After that, an invalid value was passed to the victim getter as the argument. The following line of the code was to use the getter's return value as an array's index, which will be invalid as usual. But after reloading the array and measuring the time cost, it is evident that the element with the return value as the index is in the cache. Thus, it is reasonable to take the index as the secret.

The exact measurement can be taken as the sample Meltdown attack to increase the accuracy. We can reload the array multiple times, evaluate the mean cycle number, and take the one with the least mean cycle as the secret. Nevertheless, the accuracy of the attack is not guaranteed and may fail occasionally.

3.3 Difficulties and Solutions

The biggest difficulty to demonstrate Meltdown is creating an impacted environment. After 2017, when the vulnerabilities were discovered, both hardware and software engineers have been working on them. Even before Meltdown was discovered, KAISER was invented to separate kernel mode memory address and user mode memory address.[5] If the mapping between the address and the kernel mode memory becomes invalid, it will be impossible to carry out Meltdown, because the attacker cannot even find a way to represent kernel-mode memory. Moreover, hardware patches have also been made to later CPU. Intel calls the hardware solution "protective walls", which improves both process and privilege-level separation. Some of the CPUs manufactured before 2018 and after 2008 have also been patched using Intel microcode.[6] In 2022, it is not easy to find a PC that is affected, and those servers or industrial computers are usually up-to-date and unable to virtualize old OSes. Emulators such as Qemu cannot emulate the behavior of an old CPU on a later CPU exactly, thus it fails in this demonstration. Fortunately, we managed to find an affected computer, which is not so fortunate for the owner. Spectre, however, is not hard to demonstrate, because it is still affecting most computers and can only be mitigated by software patches.[2]

4. MITIGATION

Generally speaking, these two attacks are realized through side channel technology and based on the vulnerability of modern CPU technology. It affects almost all modern devices and steals the important information of users. Because the vulnerability exists at the hardware level, it is difficult for anti-virus software and personal firewall to detect the attack, let alone protect it. However, to be more convenient and effective, we cannot change the operating mechanism of the CPU. There are various ideas for reducing the damage from these attacks, and we explain some of them.

For the Meltdown attack, we introduce KPTI, a non-trivial patch to separate page tables for user and kernel space inspired by modern operating systems (kernel page table isolation).[7]The user page table is switched to the kernel page table at the entry point of these paths whenever switching from user mode to kernel mode is necessary, such as when performing system calls or handling interrupts and exceptions. When returning from kernel mode to Before user mode, you must switch back to the user page table at all exit points. Furthermore, KPTI also cancels the configuration of the PTE corresponding to the kernel mapping as a global page attribute.

Although this patch prevents kernel memory leaks caused by malicious user processes, it forces users to patch their kernels (typically necessitating a reboot). It is currently only compatible with the most recent OS kernel versions. Furthermore, switching page tables during user/kernel crossings results in non-trivial performance overhead.

We can still not eliminate the Speculative Execution for Specter attack because it is a crucial performance-enhancing feature created by current CPUs. Thus, we begin from a different perspective: Preventing Access to Secret Data (PASD)[6]. WebKit[8] has two techniques to restrict speculatively executed code access to sensitive data. Index masking is used in place of array bounds checking in the first strategy. WebKit uses a bit mask to an array index to ensure that it is not significantly larger than the array size rather than checking that an array index is within the array's boundaries. Although masking might lead to access outside the array's boundaries, this reduces the extent of the bounds violation and stops the attacker from accessing any memory.

The second strategy protects access to pointers by doing exclusive-or on the pointer with a pseudorandom poison value. Poison protects pointers in two different ways. First, an adversary cannot use a poison pointer unless the poison value is leaked through a cache attack. Second, if a type-checked branch instruction is mispredicted, the poison value will cause the pointer-associated type to be used for another type.

5. CONCLUSION

In this paper, we explain the attack principle of Meltdown and Spectre based on two CPU mechanisms: Out-of-Order Execution and Speculative Execution, restore the attack process and propose two mitigation measures, KPTI and PASD. Overall seen from experimental results, attackers can achieve side-channel theft of classified information in modern computer systems via Meltdown and Spectre. To prevent being attacked by these two kinds of vulnerabilities, it is very necessary to install patches and update them in time.

6. REFERENCES

[1] L. Moritz, S. Michael, G. Daniel, P. Thomas, H. Werner, F. Anders, H. Jann, M. Stefan, K. Paul, G. Daniel, Y. Yuval, and H. Mike, "Meltdown: Reading Kernel Memory from User Space," 2018.

[2] A.-T. Le, T.-T. Hoang, B.-A. Dao, A. Tsukamoto, K. Suzaki, and C.-K. Pham, 'A Real-Time Cache Side-Channel Attack Detection System on RISC-V Out-of-Order Processor', IEEE Access, vol. 9, pp. 164597–164612, 2021, doi: 10.1109/ACCESS.2021.3134256.

[3] P. Kocher et al., 'Spectre Attacks: Exploiting Speculative Execution'. arXiv, Jan. 03, 2018. Accessed: Jul. 27, 2022. [Online]. Available: http://arxiv.org/abs/1801.01203

[4] W. Du, "Meltdown Attack Lab," Computer Security: A Hands-on Approach, 2018.

[5] J. Corbet. "KAISER: hiding the kernel from user space," Jul. 27, 2022; https://lwn.net/Articles/738975/.

[6] B. Krzanich, "Advancing Security at the Silicon Level," Hardware-based Protection Coming to Data Center and PC Products Later this Year, Intel, 2018.

[7] Corbet, J., 2017. The current state of kernel page-table isolation. https://lwn.net/ARTICLEs/741878/ Accessed Sep 26, 2020.Google Scholar

[8] F. Pizlo, "What Spectre and Meltdown mean for WebKit," Jan. 2018.Available:https://webkit.org/blog/8048/what-spectreand-meltdown-mean-for-webkit/

An empirical study of open-source black-box web application scanners

Chaosong Zhang*
Chaosong.Zhang20@student.xjtlu.edu.cn
Xi'an Jiaotong-Liverpool University
Suzhou, China

Jiayuan Liang*
11910504@mail.sustech.edu.cn
Southern University of Science and Technology
Shenzhen, China

Shengkai Ma*
878916392@qq.com
Huazhong University of Science and Technology
Wuhan, China

Sihan Guo*
1367735726@qq.com
Xi'an Jiaotong University
Xi'an, China

Runhe Ma*
amour_mo@126.com
Qingdao University
Shandong, China

Abstract

Web applications have had significant influence on many aspects of our society. However, security related vulnerabilities are common among them, which can seriously affect the users' experience. In some cases, these vulnerabilities may lead to information leakage or even financial crimes. Web application vulnerability scanners (WAVS) are a set of automated tools that can exam web applications for vulnerabilities such as SQL injections (SQLI), cross site scripting (XSS) etc. In this paper, we aim to evaluate some of the common open-source black-box WAVS to see their effectiveness in vulnerabilities detecting.

Keywords: black-box, web applications, vulnerability, open-source

1 Introduction

Web applications are playing a crucial role in many aspects of our society. However, security vulnerabilities are commonly found in them[5]. Once exploited by malicious attackers, these vulnerabilities can lead to information leakage or even financial crimes. To protect the web applications from attacks, multiple web application vulnerability scanners have been developed.

Web application vulnerability scanners are a set of automated tools that can exam web applications for vulnerabilities such as SQL injections, cross site scripting etc. In this paper, we will evaluate some of the common open-source black-box WAVS based on the number of vulnerabilities they detect and severity levels of them. Our data are publicly available[1].

Our paper is structured as follow: In section 2, we will introduce our research background by explaining the basic concepts related to our topic. In section 3, the design of our evaluation will be illustrated including our evaluation methodology, metrics, tool and benchmark selection. In section 4, we will demonstrate the result of our evaluation and in section 5 draw a conclusion from our result.

*Equal contribution.

[1]https://github.com/JustinLiang522/DOTA-Group2-Data

2 Research Background

2.1 Web Applications

Web applications, also known as web apps, are software that run on web browsers. They are delivered through the World Wide Web to users with internet connection and can responds to dynamic web page requests over HTTP. Web apps are usually developed with programming language such as JavaScript, PHP, Ruby etc.

Figure 1, present a simplified view of how a web app function. A web app will provide contents to web clients in respond to their HTTP requests. It also establish a connection with the Database server to retrieve requested files.

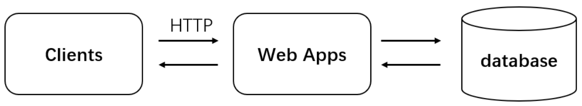


Figure 1. Web Apps

2.2 Web Application Vulnerability Scanners

WAVS are defined as automated programs that examine web applications for security vulnerabilities[1]. These programs can analyze a vulnerable website by performing penetration tests on it.

The general workflow of WAVS can be summarized as follow: First, the user enter a URL to be tested; Then the WAVS will analysis all the pages of the URL and send GET or POST requests to the target pages. Meanwhile, it will analyze the received response to see if there exists any vulnerability. After analyzing all the result, it will generate a report with all the detected vulnerabilities with their types and severity levels.

There are commercial and open source WAVS. But in this paper, we only focus on the open source tools.

2.3 Penetration Test

A penetration test, also known as ethical hacking, is an authorized simulated cyberattack on a computer system, performed to evaluate the security of the system. The test is performed to identify vulnerabilities, including the potential for unauthorized parties to gain access to the system's features and data, as well as strengths, enabling a full risk assessment to be completed.

The process typically identifies the target systems and a particular goal, then reviews available information and undertakes various means to attain that goal. A penetration test target may be a white box (about which background and system information are provided in advance to the tester) or a black box (about which only basic information—if any—other than the company name is provided). A penetration test can help identify a system's vulnerabilities to attack and estimate how vulnerable it is.

2.4 Vulnerability

A vulnerability in cyber security refers to any weakness in an information system, system processes, or internal controls of an organization. Vulnerabilities in web applications are often target of malicious attacks.

Table 1 list the ten most common web application vulnerabilities in 2021[3] reported by OWASP which includes Broken Access Control, Cryptographic Failures, Injection etc.

Ranking	Vulnerability
1	Broken Access Control
2	Cryptographic Failures
3	Injection
4	Insecure Design
5	Security Misconfiguration
6	Vulnerable and Outdated Components
7	Identification and Authentication Failures
8	Software and Data Integrity Failures
9	Security Logging and Monitoring Failures
10	Server-Side Request Forgery (SSRF)

Table 1. Top 10 vulnerabilities

2.5 Severity Level

Most WAVS classify vulnerabilities into four categories: High, Medium, Low, and Informational.

High level vulnerability is a type of vulnerability that once exploited will allow the attacker full control over the web application and server, putting clients information severely in danger. SQLI and XSS are vulnerabilities of the level.

Medium level vulnerability is a type of vulnerability that once exploited will allows attackers to access a logged-in user account to view sensitive content. It gives attackers access to data that enables them to exploit other flaws or gain a deeper understanding of the system to improve their attacks. An example of a medium severity vulnerability is open redirection, which enables an attacker to redirect a user to a malicious website. If a scanner identifies vulnerabilities of medium severity, they should be fixed as soon as possible.

Low level vulnerabilities are those that have little or no effect and cannot be used by an attacker. A low severity vulnerability can be cookies that are not HttpOnly marked. Cookies that have been marked as HttpOnly cannot be accessed by client-side scripts, adding another level of defense against XSS attacks. If time and money permit, it is worthwhile to investigate and patch low severity vulnerabilities.

Informational level alters can be considered as notifications that provide information about the web applications. No action is needed for them.

3 Evaluation Design

3.1 Evaluation Methodology

Figure 2 show the workflow of our evaluation method. First, we will collect the existing free or open-source WAVS and use them to scan the selected web applications. Then, we will manually validate the reports and compare the result. This step is crucial since different tools adopt different way of analyzing their result, for instance, how they categorize and count the vulnerabilities and how they determine the severity level can be quite different. In the end, we will draw a conclusion from our comparison.

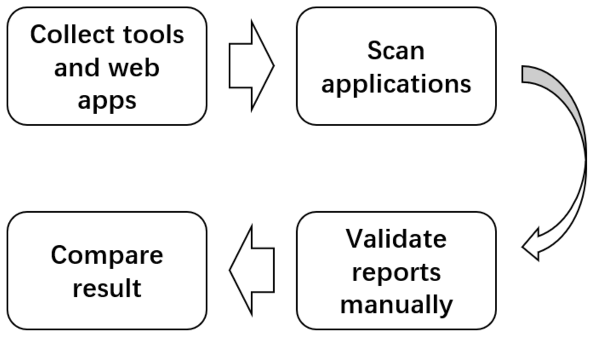


Figure 2. Research Method

3.2 Tools and Websites Selection

Table 2 shows the 4 web applications we used as targets of our scanners and Table 3 shows the 3 popular open-source black-box WAVS that we have used for evaluation. Our web

index	Web Address	Description
W1	testaspnet.vulnweb.com	A news blog website created by Acunetix organization as a testing application for scanners, written in asp.net language.
W2	testphp.vulnweb.com	An online shopping web application created by Acunetix as a testing application for scanners, written in PHP language.
W3	zero.webappsecurity.com	An online banking web application created by Hewlett-Packard (HP) to test web vulnerability scanners.
W4	testhtml5.vulnweb.com	An online social networking application created by Acunetix to test web application scanners, written in HTML5 language.

Table 2. Vulnerable Web Applications

Scanner	Vendor	Version	Platform
OWASP ZAP	OWASP	2.11.1	Windows, Linux, OS X
Wapiti	Nicolas Surribas	3.1.3	Linux
Vega	Subgraph	1.0	Windows, Linux, OS X

Table 3. Scanners

apps and tools are collected from previous work[4][2] and GitHub.

3.2.1 OWASP ZAP. is a free and open-source web application vulnerability scanner with graphic user interface (GUI). It is one of the most active Open Web Application Security Project (OWASP) projects intended to be used by both those new to application security as well as professional penetration testers. The source code of this tool is available on GitHub[8].

3.2.2 Wapiti. is a free and open-source web application vulnerability scanner written in python3. It can generate well-organized report of the detected vulnerability in multiple format such as html. The source code of this tool is available on GitHub[7].

3.2.3 Vega. is a free and open source web security scanner and web security testing platform to test the security of web applications developed by Subgraph in Montreal. It can find and validate SQL Injection, Cross-Site Scripting, and other vulnerabilities. It is written in Java, GUI based, and runs on Linux, OS X, and Windows. The source code of Vega is available on GitHub[6].

3.3 Evaluation Metric

To evaluate the selected WAVS, we have formulated the following research questions.

RQ1. How many vulnerabilities can the WAVS detect in total?
RQ2. How many vulnerabilities can the WAVS detect of different security levels?
RQ3. How many vulnerabilities can the WAVS detect of common categories?

4 Evaluation Result

4.1 RQ1. How many vulnerabilities can the WAVS detect in total?

From the result shown in Table 4 and Figure 3, we can see that ZAP detected the most vulnerabilities in these web apps. Followed by Wapiti, which detect 148 vulnerability in total, about twice the number of that detected by Vega.

Scanners	Web Applications				
	W1	W2	W3	W4	Total
OWASP ZAP	183	258	40	32	513
Wapiti	96	41	5	6	148
Vega	24	11	13	22	70

Table 4. Total number of detected vulnerabilities

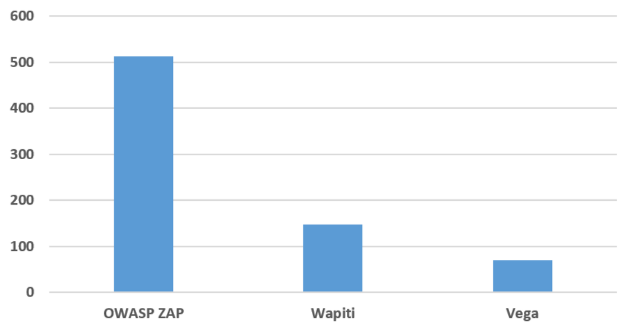


Figure 3. Number of detected vulnerabilities

Interestingly, what accounts for the difference in the result of these tool is not merely their efficiency, but also the strategy they use to count and categorize vulnerabilities. For instance, ZAP and Wapiti count the vulnerabilities based on both the URL and parameters while Vega only based on the URL.

4.2 RQ2. How many vulnerabilities can the WAVS detect of different security levels?

From the result shown in Table 5 and Figure 4. We can see that Medium and Low level vulnerabilities take up a large portion of the vulnerabilities that detected by ZAP. Wapiti does not reveal informational vulnerability in web applications but it detect the most High level vulnerability among these tools. The number of High level vulnerabilities detected by Vega are close to that of ZAP, but the detected Medium and Low level vulnerabilities are much less.

Scanners	Severity level			
	High	Medium	Low	Info
OWASP ZAP	21	95	353	44
Wapiti	91	25	32	0
Vega	20	4	8	38

Table 5. Vulnerabilities based on severity levels

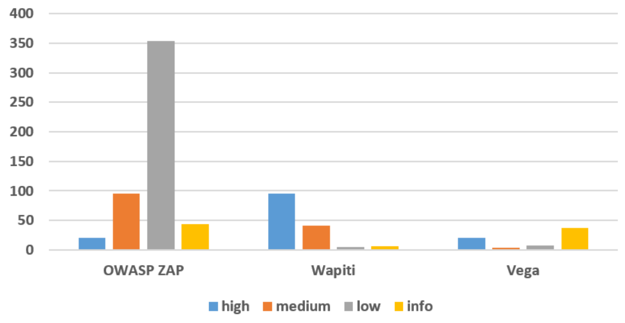


Figure 4. Vulnerabilities based on severity levels

4.3 RQ3. How many vulnerabilities can the WAVS detect of common categories?

To answer RQ3, we focus on three common types of vulnerabilities: SQL injection (SQLI), stored cross site scripting (SXSS) and reflected cross site scripting (RXSS). From the result shown in Table 6 and Figure 5, we can see that Vega did not detect any form of XSS vulnerabilities while wapiti detected much more vulnerabilities of these two type than the others. This drastic difference is due mainly to the way these tools count the number of detected vulnerabilities, for example, Vega count vulnerabilities based on the attacked URL while Wapiti distinguish the vulnerabilities by the parameters as well, which can account for the large number of vulnerabilities it found. But regardless of the difference in their counting method, wapiti still report the most of these three common vulnerabilities even if we count them only based on URL.

Scanners	Vulnerabilities		
	SQLI	SXSS	RXSS
OWASP ZAP	2	1	3
Wapiti	29	60	25
Vega	3	0	0

Table 6. Detected common vulnerabilities

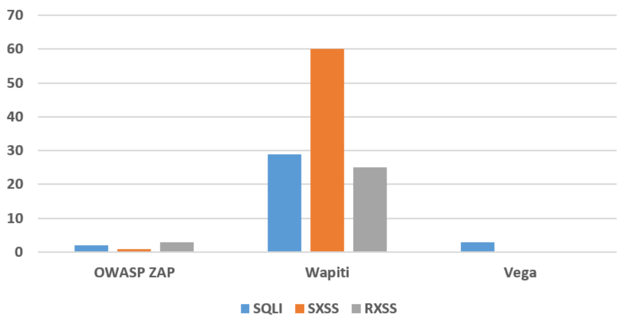


Figure 5. Detected common vulnerabilities

5 Conclusion

In this work, we have evaluated 3 open-source black-box web application vulnerability scanners, based on the number of vulnerabilities they detected and the severity levels of them. We found that, among the evaluated tools, Wapiti detected the largest number of high level vulnerabilities and ZAP is capable of revealing the most medium an low level ones even if they adopt different counting and categorizing strategies. Hopefully, our work can help improve the efficiency of the evaluated scanners.

References

[1] Elizabeth Fong and Vadim Okun. 2007. Web application scanners: definitions and functions. In *2007 40th Annual Hawaii International Conference on System Sciences (HICSS'07)*. IEEE, 280b–280b.
[2] SE Idrissi, N Berbiche, F Guerouate, and M Shibi. 2017. Performance evaluation of web application security scanners for prevention and protection against vulnerabilities. *International Journal of Applied Engineering Research* 12, 21 (2017), 11068–11076.
[3] OWASP. 2021. *OWASP Top 10:2021*. https://owasp.org/Top10/
[4] Malik Qasaimeh, A Shamlawi, and Tariq Khairallah. 2018. Black box evaluation of web application scanners: Standards mapping approach. *Journal of Theoretical and Applied Information Technology* 96, 14 (2018), 4584–4596.
[5] Whitehat Security. 2015. *Web Security Statistics Report 2015*. https://info.whitehatsec.com/rs/whitehatsecurity/images/2015-Stats-Report.pdf
[6] Vega. 2016. *Subgraph Vega*. https://github.com/subgraph/Vega
[7] Wapiti. 2022. *Wapiti*. https://github.com/wapiti-scanner/wapiti
[8] zaproxy. 2022. *OWASP ZAP*. https://github.com/zaproxy/zaproxy

A Static Analysis Method for Detecting Buffer Overflow Vulnerabilities

Zhang Shuhao [*]
Nankai University
shuhao-zhang@outlook.com

Qin Jianxing [†]
Shanghai Jiao Tong University
qdelta@sjtu.edu.cn

Qu Shaobo [‡]
Huazhong University of Science and Technology
commcheck396@gmail.com

Hong Yun [§]
Xi’an Jiao Tong University
yoyoball@stu.xjtu.edu.cn

ABSTRACT

Buffer overflows are a weakness that is commonly found in some languages, such as C and Java. Any code without a strict boundary limit can pose a potential overflow risk and it’s easy to be unaware of them. We demonstrate an approach to verification of C-like programs using analysis of the source code of programs. The approach applies a formal definition of the syntax and semantics of the subset of C and makes use of symbolic execution. Our approach is illustrated to capture all overflows of the source code written in our object language.

Keywords

buffer overflow, vulnerability in programs, static analysis, symbolic execution

1. INTRODUCTION

Buffer overflow is a typical vulnerability in programs. Many attacks on Microsoft systems are based on various buffer overflow problems. However, in some commonly used languages like C or Java, these problems will not be pointed out by the compiler.

In this paper, we will use symbolic execution to detect buffer overflow problems in small programs which are written in the subset of C and develop an analyzing application with a GUI.

[*]Zhang wrote the code of our core part and helped with the revision of our paper.

[†]Qin played the role as a leader. He provides most of the theoretical knowledge, wrote the code of our core part and helped with paper writing.

[‡]Qu wrote the code of our GUI and made our poster.

[§]I wrote most of this paper and made our video.

2. BACKGROUND

2.1 A glance at Buffer overflow attack

As its name suggests, buffer overflow occurs when a program attempts to write more data to a fixed-length block of memory or buffer than the buffer is allocated to hold[2]. Once an overflow occurs, the extra data will overwrite the data value in adjacent memory addresses of the destination buffer. This consequence can be used to crash a process or modify its internal variables. The attack that is based on this consequence is called buffer overflow.

This kind of attack can be fatal because the original data in the buffer includes the return pointer of the exploited function. The attacker can use the extra data to modify these pointers and modify the address to which the process should go next according to their own wishes. For example, they can make it point to a dangerous attack program and easily complete their attack.

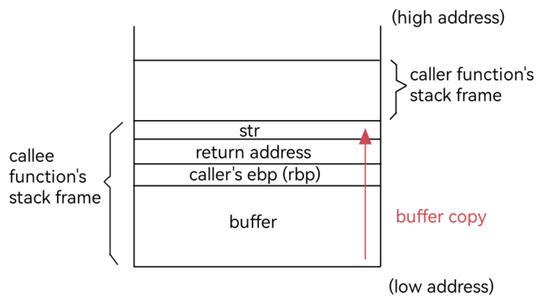


Figure 1: buffer overflow occurs when copying string str

Furthermore, buffer overflow attack ranks high in Common Weakness Enumeration (CWE)[2]. This sort of vulnerability can occur easily in programs without sufficient bounds checking. The preventing work can be sophisticated, especially when the occurrence of this kind of vulnerable problem will not always be warned by the compiler. As a result, an assistant tool with a function of checking potential overflow risk can be significant.

2.2 Symbolic execution overview

Symbolic execution[1] is a way of executing a program abstractly. This type of execution treats the inputs of the program symbolically and focuses on execution paths through the code. To be more specific, the interpreter will assume symbolic values for inputs rather than obtaining actual inputs as normal execution of the program would[4], and the result of a symbolic execution should be expressed in terms of symbolic constants that represent the input values.

By doing an abstract execution, multiple possible inputs of the program that share a particular path can be covered[1]. In addition, symbolic execution also has another strength, that is, it can avoid reporting false warnings because each of all these errors represents a particular path through the program. This provides us an idea of checking potential buffer overflows. For any program statement that has the possibility of occurring buffer overflow problems, we can check the value range of those involved variables to verify whether they are satisfied the boundary constraints. If any of these variables fail to meet the condition, we can say that a buffer overflow problem occurs.

3. DEFINITION OF LANGUAGE

To simplify the process, we set the language of the source code to a subset of C. For the convenience of illustrating our ideas, we will use a less complicated language (we may call the language SymC--), which is semantically a subset of C, to describe our object language. Here are the detailed definitions.

Expression:

$e ::=$	n	integer literal
$\mid$	x	name
$\mid$	$e\ \mathit{bop}\ e$	binary operation
$\mid$	$\mathit{uop}\ e$	unary operation
$\mid$	$\texttt{alloc}(e)$	memory allocation

Statement:

$s ::=$	$\texttt{skip}$	empty statement
$\mid$	$\texttt{declint}\ x$	integer declaration
$\mid$	$\texttt{declptr}\ x$	pointer declaration
$\mid$	$x \leftarrow e$	assignment
$\mid$	$\texttt{touch}\ e[e]$	memory access
$\mid$	$s; s$	sequence
$\mid$	$\texttt{if}\ e\ \texttt{then}\ s\ \texttt{else}\ s$	conditional

Values:

$v ::=$	$\texttt{int}\ v_i$	integer
$\mid$	$\texttt{ptr}\ v_i\ v_i$	pointer with bounds
$v_i ::=$	n	concrete integer
$\mid$	x	symbolic integer
$\mid$	$\mathit{uop}\ v_i$	unary operation
$\mid$	$v_i\ \mathit{bop}\ v_i$	binary operation
$\mid$	$\texttt{ite}\ a\ v_i\ v_i$	conditional

Assertions:

$a ::=$	$v_i\ \mathit{relop}\ v_i$	integer relation
$\mid$	$\neg a$	negation
$\mid$	$a \wedge a$	conjunction
$\mid$	$a \vee a$	disjunction

It is also necessary to emphasize that this demo language sharing the same semantics with our source code language won't be involved in our code implementation. The purpose of developing such language is just for the convenience of illustration.

4. PROCESS OF ANALYSIS

The general process of analysis involves several parts.

The first part includes two preparation tasks. One is to check whether the source code can be compiled successfully. Another is to do an initial analysis with a parser and gain an abstract syntax tree of the code. The purpose of this step is to transform the code into an abstract form for the subsequent analysis.

In the second part, we will use symbolic execution to extract useful information from the result of the first step, and generate counterexamples of the variables involved in statements which have the risk of occurring buffer overflow.

Next, a SMT-solver will be used to check whether any of these counter examples can be satisfied. We will have a deeper discussion on how this solver can be used in part 4.3. With the feedback from the solver, we can locate the potential buffer overflows in the source code.

The last part of our analysis is to print the errors discovered to a GUI.

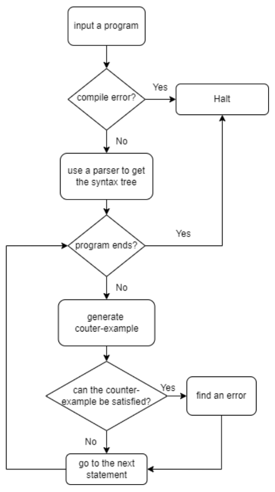


Figure 2: The whole process of the analysis[3]

Now, we will explain each part respectively.

4.1 Preparation

The first part of the preparation is to get the source code the user input and compile it. Our goal is to uncover those potential risks which cannot be warned by the compiler and our target source code should be those which can be compiled. If an error occurs, we will print the error information to the output box and halt the whole process.

Only if a source code can be compiled, will it be given to a parser to generate an abstract syntax tree. We use pycparser[1] here, which is a parser of C and written in Python. It can be replaced by any parser of C. The output will then be the input of the next part.

4.2 From program to counterexamples

The goal of this part is to generate counterexamples. In this part, we will use symbolic execution.

First, we will give a more specific illustration of this method.

When we do symbolic execution, we are actually executing a path through the source code. During this process, we will track the values of variables and the conditions that need to be satisfied for a specific path. It's necessary to mention that the values that we will keep track of are all expressed in terms of symbolic values. For example, if we have a user input variable a and execute $b = a + 1$, then the values we track should be

$$a \mapsto \alpha, b \mapsto \alpha + 1$$

In this example, α is a symbolic value, and $a \mapsto \alpha, b \mapsto \alpha + 1$ is what we will keep track of. We can use symbolic environment E as a collective name for the mapping from name to value that we keep track of.

To get a clearer picture of the process, we can take the code below as an example.

```
void foo(int *arr, int n, int head) {
    if (head > 0) {
        print(arr[0]);
    } else {
        print(arr[n]);
    }
}
```

In this function, we will track the value of arr, n, and $head$. Their symbolic constants are $ptr\ 0\ h', n'$ and $head'$. The symbolic environment E looks like this.

$$E = \{arr \mapsto \texttt{ptr}\ 0\ h',\ n \mapsto n',\ head \mapsto head'\}$$

When we meet conditional statement, we will track the condition for each branch. When we come to the first branch, the path condition P we will be tracking is $head' > 0$. For the second branch, the path condition P is $head' \leq 0$. The symbolic environment E is not updated in this function.

If there are any assumptions in the source code, it should be described by symbolic environment E and path condition P.

[1]You can find more information about pycparser at https://github.com/eliben/pycparser

```
void foo(int *arr, int n, int head) {
    ASSUME(n > 0, capacity(arr) >= n);
    if (head != 0) {
        print(arr[0]);
    } else {
        print(arr[n]);
    }
}
```

For example, if we add an assumption at the beginning of the function, the initial E and P should be look like this when after executing the assumption statement.

$$E = \{arr \mapsto ptr\ 0\ h',\ n \mapsto n',\ head \mapsto head'\}$$
$$P = \{n' > 0,\ h' \geq n'\}$$

Now we can introduce our idea of generating counterexamples with symbolic execution. For each memory access, the index should be always bounded in the range of the accessed array. This is our bounded requirement Q. Also, we have a path condition P which should be true when the program executes the corresponding statement. So it has to be guaranteed that $P \to Q$ in all cases, which implies $P \land \neg Q$ should not be satisfied. So we call $P \land \neg Q$ a counterexample.

Evaluation rules in the form of $E; e \Downarrow v$. It means expression e is evaluated to value v in environment E.

$$\frac{}{E; n \Downarrow \texttt{int}\ n} \qquad \frac{}{E; x \Downarrow E(x)}$$

$$\frac{E; e \Downarrow \texttt{int}\ v}{E; uop\ e \Downarrow \texttt{int}\ (uop\ v)} \qquad \frac{E; e_1 \Downarrow \texttt{int}\ v_1 \quad E; e_2 \Downarrow \texttt{int}\ v_2}{E; e_1\ bop\ e_2 \Downarrow \texttt{int}\ (v_1\ bop\ v_2)}$$

$$\frac{E; e_1 \Downarrow \texttt{ptr}\ l\ h \quad E; e_2 \Downarrow \texttt{int}\ v}{E; e_1 + e_2 \Downarrow \texttt{ptr}\ (l - v)\ (h - v)} \qquad \frac{E; e_1 \Downarrow \texttt{int}\ v \quad E; e_2 \Downarrow \texttt{ptr}\ l\ h}{E; e_1 + e_2 \Downarrow \texttt{ptr}\ (l - v)\ (h - v)}$$

$$\frac{E; e_1 \Downarrow \texttt{ptr}\ l\ h \quad E; e_2 \Downarrow \texttt{int}\ v}{E; e_1 - e_2 \Downarrow \texttt{ptr}\ (l + v)\ (h + v)}$$

$$\frac{E; e \Downarrow \texttt{int}\ v}{E; \texttt{alloc}(e) \Downarrow \texttt{ptr}\ 0\ v}$$

Execution rules in the form of $E, P; s \Downarrow E'; C$. It means given environment E and path condition P, after executing the statement s the environment will be updated to E' and counter examples C will be generated.

Path condition P is an assertion. C is the set of generated assertions that should not be satisfied.

$$\frac{}{E; n \Downarrow \texttt{int}\ n} \qquad \frac{}{E; x \Downarrow E(x)}$$

$$\frac{}{E, P; \texttt{declint}\ x \Downarrow E \cup \{x \mapsto \texttt{int}\ x'\}; \{\}}$$

$$\frac{}{E, P; \texttt{declptr}\ x \Downarrow E \cup \{x \mapsto \texttt{ptr}\ 0\ 0\}; \{\}}$$

$$\frac{E; e \Downarrow v}{E, P; x \leftarrow e \Downarrow E[x \mapsto v]; \{\}}$$

$$\frac{E, e_1 \Downarrow \texttt{ptr}\ l\ h \quad E, e_2 \Downarrow \texttt{int}\ v}{E, P; \texttt{touch}\ e_1[e_2] \Downarrow E; \{P \land (l > v \lor v \geq h)\}}$$

$$\frac{E, P; s_1 \Downarrow E'; C_1 \quad E', P; s_2 \Downarrow E''; C_2}{E, P; s_1; s_2 \Downarrow E''; C_1 \cup C_2}$$

$$\frac{E; e \Downarrow \texttt{int}\ v \quad E, P \land v \neq 0; s_1 \Downarrow E_1; C_1 \quad E, P \land v = 0; s_2 \Downarrow E_2; C_2}{E, P; \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \Downarrow \text{merge}(v, E_1, E_2); C_1 \cup C_2}$$

$\text{merge}(v, E_1, E_2) = E$ means $\forall x \in E_1, E_2$:

$$E_1(x) = \texttt{int}\ v_1, E_2(x) = \texttt{int}\ v_2$$
$$\to E(x) = \texttt{int}\ (\texttt{ite}\ (v = 0)\ v_2\ v_1)$$

$$E_1(x) = \texttt{ptr}\ l_1\ h_1, E_2(x) = \texttt{ptr}\ l_2\ h_2$$
$$\to E(x) = \texttt{ptr}\ (\texttt{ite}\ (v = 0)\ l_2\ l_1)\ (\texttt{ite}\ (v = 0)\ h_2\ h_1)$$

else error.

To get an intuition for what we can get from the process, we can take the code above (function $foo()$ with assumption) as an example.

After execution, the generated counterexamples will be

$$n' > 0 \land h' \geq n' \land head' \neq 0 \land (0 > 0 \lor 0 \geq h)$$
$$n' > 0 \land h' \geq n' \land head' = 0 \land (0 > n' \lor n' \geq h)$$

With the rules, we can write a program to realize this function.

4.3 Counterexample analysis

The goal for this step is to check whether any of the counterexamples we get from 4.2 can be satisfied.

Suppose that we now get a set of counterexamples C_i, $i = 1...n$. We should check each of them. If a particular counterexample can be satisfied, the line where we generate it could have overflow risk. Now the question has transformed to whether there exists a mapping from symbolic values to concrete values, such that $C_i = true$.

So far, we've transformed our original problem into a satisfiability problem. We can use SMT-solver z3[2] to solve it and find out the counterexamples which can be satisfied. The statement where we generate them are the errors we found.

5. GUI AND RUNNING RESULTS

After previous work, we have located the errors. The last stage of the analysis is to print these errors. In this part, we will provide an example code and its running results.

Our GUI is split into two parts. Users can type their source code waiting for analyzing to the left box and get the result from the box on the right.

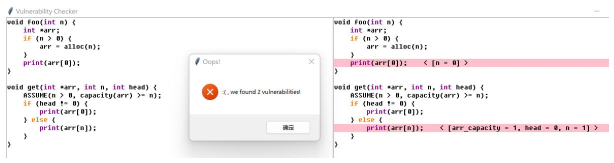
Target source code

```
void foo(int n) {
    int *arr;
    if (n > 0) {
        arr = alloc(n);
    }
    print(arr[0]);
}

void get(int *arr, int n, int head) {
    ASSUME(n > 0, capacity(arr) >= n);
    if (head != 0) {
        print(arr[0]);
    } else {
        print(arr[n]);
    }
}
```

[2] You can find more information about z3 at https://github.com/Z3Prover/z3/wiki#background

The running result is like this.



```
void foo(int n) {
    int *arr;
    if (n > 0) {
        arr = alloc(n);
    }
    print(arr[0]);
}

void get(int *arr, int n, int head) {
    ASSUME(n > 0, capacity(arr) >= n);
    if (head != 0) {
        print(arr[0]);
    } else {
        print(arr[n]);
    }
}
```

:(, we found 2 vulnerabilities!

```
void foo(int n) {
    int *arr;
    if (n > 0) {
        arr = alloc(n);
    }
    print(arr[0]);    < [n = 0] >
}

void get(int *arr, int n, int head) {
    ASSUME(n > 0, capacity(arr) >= n);
    if (head != 0) {
        print(arr[0]);
    } else {
        print(arr[n]);    < [arr_capacity = 1, head = 0, n = 1] >
    }
}
```

Figure 3: Running result of the example code above.

Our analyser returns two errors. They're highlighted and provided with counter examples that can be satisfied.

6. RELATED WORKS AND COMPARISON

In the method we mentioned above, we're actually doing static analysis. There's also another way to test vulnerability of programs, which is called dynamic analysis. Unlike static analysis, dynamic analysis evaluates the program by executing data in real-time. The input cases can be generated by a program using random algorithm, or be typed in by the tester.

What these two kinds of methods have in common is that they're both trying to find counterexamples. Though dynamic analysis can prove a program has overflows, it can be overwhelmed when faced with a correct program. The method we mentioned, on the other hand, uses static analysis and attempts to find counterexamples in a more theoretical way, which allows it to prove the non-existence of counterexamples.

Besides, our method has another advantage over dynamic analysis in efficiency. While a particular input case can only find out some of the buffer overflows, symbolic execution can check every possible path in the program. This allows us to find all the errors with only one execution instead of testing endlessly.

7. LIMITATIONS

Our project is a very basic prototype. It has several known limitations shown in our semantics model.

- The values in the array are discarded.
- Loops are not supported in the language.
- The program has only one exit point at the end.

8. CONCLUSION

From the represented results we can see that our method has advantages of precision, efficiency and automation. However, there are also some shortcomings. First is that the solver we use in counterexamples analysis (4.3) is unstable. The reason is that the satisfiability problem we are trying to solve is a NP-hard problem. The second shortage is that since the process gives another program as the input of our program, it can't handle any program according to the halting problem.

However, in most of the cases, our method can perform well and give a correct result quickly. So we can still say that the advantages outweigh the disadvantages and this method can be of help in testing programs for vulnerabilities.

9. ACKNOWLEDGEMENT

I would like to express my very great appreciation to Professor Hugh Anderson and his teaching assistant Harish Venkatesan for their valuable and constructive suggestions during the planning and development of our project.

I would also like to thank my group-mates. They did the really hard work of our coding part and gave me useful suggestions in paper writing.

References

[1] J Aldrich. *Symbolic Execution (Program Analysis lecture notes - Spring 2019)*. https://www.cs.cmu.edu/~aldrich/courses/17-355-19sp/notes/notes14-symbolic-execution.pdf. 2019.

[2] Michael Cobb. *buffer overflow*. https://www.techtarget.com/searchsecurity/definition/buffer-overflow.

[3] Wenliang Du. *Computer Security:A Hands-on Approach*. Wenliang Du, 2019.

[4] Wikipedia. *buffer overflow*. https://en.wikipedia.org/wiki/Symbolic_execution.

Audio Steganography and Watermark

Zhang Jingmiao
University of Science and Technology of China
nanshan@mail.ustc.edu.cn

Wei Panyue
Huazhong University of Science and Technology
weipp7@gmail.com

Wang Junchao
Sichuan University
junchaowang613@gmail.com

Wei Xinyue
Xi'an Jiaotong University
wxyeipai@stu.xjtu.edu.cn

ABSTRACT

This paper proposes methods to apply LSB, DWT, DWT-LSB and DCT algorithms to audio digital watermarking and explores their robustness respectively. Among them, the LSB method is based on the replacement of the least significant bits, the DWT is based on the Discrete Wavelet Transform, the DWT-LSB is the combination of DWT and LSB, and the DCT is based on the Discrete Cosine Transform. To evaluate their robustness, we tested their performance for several audio manipulations, and overall they sometimes perform well and sometimes perform poorly. We also propose the whole process of adding, extracting, and verifying digital watermarks in practical application scenarios, and LSB watermarking can be competent in this scenario.

Keywords

Watermarking, Cryptography, Copyright Protection, Robustness

1. INTRODUCTION

Digital watermarking technology is an information hiding technology. The so-called audio digital watermarking algorithm is to embed a digital watermark into an audio file (such as .wav, .mp3, .avi, etc.) through a watermark embedding algorithm, but it has no effect on the original sound quality of the audio file or the human ear can't feel its effect. On the contrary, through the watermark extraction algorithm, the audio digital watermark is completely extracted from the audio host file. The watermark must be robust to attacks and other types of distortion to prevent tampering and forgery. Typical attacks include adding noise, data compression, filtering, resampling, A/D-D/A conversion, statistical attacks, etc.

The digital watermark can be identified and recognized by the producer. Through the information hidden in the carrier, it can achieve the purpose of confirming the content creator, buyer, transmitting secret information or judging whether the carrier has been tampered with. Digital watermarking is an effective way to protect the information security, and realize anti-counterfeiting traceability, and copyright protection. It is an important branch and research direction in the field of information hiding technology.

We apply LSB, DWT, DWT-LSB and DCT methods to audio watermarking, respectively, and discuss their robustness and applicable scenarios. Since they are not guaranteed to be completely indestructible and tampered with, we also propose an application method in a double-ended scenario to ensure audio integrity and copyright information.

2. PREVIOUS WORKS

There is a rich body of literature on audio digital watermarking technology. As early as 1996, Laurence Boney et al proposed a digital watermark for audio signals [1]. Ingemar J. Cox et al proposed a method for inserting watermarks in the frequency domain [2]. The combination of the DCT and LSB method is regarded as a fragile watermark [3]. Lee, S. J. and Jung, S. H. introduced the wide application of DWT in audio watermarking [4].

3. OUTLINE

We introduce the LSB, DWT, DWT-LSB and DCT algorithms in §4 and propose methods to evaluate their robustness. In §5 we introduce the practical application process and robustness analysis of the four algorithms. §6 proposes a secure watermarking and verification scenario. Finally, §7 summarizes the above work.

4. AUDIO WATERMARKING ALGORITHMS

4.1 Least Significant Bits

The basic idea of LSB replacement steganography is to replace the lowest bit of the carrier image with the secret information to be embedded. When the information is embedded, the probability of the lowest bit being changed is at most 50%. Even so, the LSB replacement steganography only introduces very little noise in the original image, which is inaudible. The audio file is encoded into an 8-column binary matrix, and changing its lowest bit has little effect on the sound quality. Therefore, embedding the binary code of the watermark file in these positions can achieve the effect of steganography.

LSB replacement steganography can be described as follows: First, the following notation is introduced: c represents the original audio (carrier) in the embedding process, which can be represented by a sequence c_i of length $l(c)$; s represents the audio after embedding the watermark, which can also be regarded as a sequence s_i of length $l(c)$, $1 \leq i \leq l(c)$; m represents the secret message to be embedded, which is a sequence of m_i with a length of $l(m)$, $l(m) \leq l(c)$, generally speaking, $m_i \in \{0, 1\}$. j represents the index value of audio. j_i indicates the order of index values. c_{ji} represents the j_i carrier element. k represents the steganography key.

The embedding and extracting process of LSB replacement steganography is as Algorithm 1:

Algorithm 1 LSB Watermarking Algorithm

```
1: Select the first $l(m)$ replaceable bits in the audio.
2: each bit selected, if its LSB is the same as the informa-
   tion bit to be embedded, do not change it; otherwise,
   proceed to the next step;
3: Replace the LSB of the original positions with the wa-
   termark information bits, while the high 7 bits remain
   unchanged, and the modified image is $s$.
4: Extract the LSB of the first $l(m)$ audio bits and arrange
   them to form secret information $m$.
```

4.2 Discrete Wavelet Transform

Discrete Wavelet Transform (DWT), receives a discrete signal as input $x(n)$ and outputs another discrete signal. The general idea of DWT is to separate the high-frequency and low-frequency signals with two filters, the higher filter and the lower filter. After the separation process, the array $x[n]$ which has a length of N, is transformed into two arrays, x_H and x_L, whose lengths are both $N/2$. X_H represents the high-frequency signals, and x_L refers to the low-frequency signals.

There are two functions $h(x)$ and $g(x)$ used as filters. From $x[n], n = 1, 2, \ldots, N$, we calculate the low-frequency signal and the high-frequency signal in the following way:

$$x_{1,L}[n] = \sum_{k=0}^{K-1} x[2n-k]g[k] \tag{1}$$

$$x_{1,H}[n] = \sum_{k=0}^{K-1} x[2n-k]h[k] \tag{2}$$

This process can continue to extend, as shown in Figure 1.

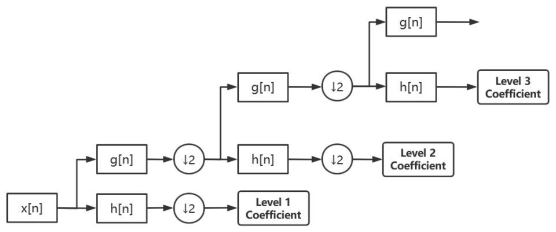


Figure 1: Extension of DWT Signal

In theory, the process can extend indefinitely. While in practice, the number of filtered layers can be determined depending on the specific case. In our project, the number of filtered layers is 2, for if we use more layers, the fewer information can be hidden, but if we use fewer layers it can store more information but may influence the primary audio. Notably, this process is theoretically fully recoverable, so when we get the array after DWT, we can recover the primary array.

Audio can be converted to an integer array, which is the 1-dimension discrete signal. We use a short integer array, in which every value is between 0 and 256, to present a grey image. After we use 2-level-DWT to the audio array, we get high-frequency signals in the second level. If we replace this array partly with the short integer array which stands for the picture and then use iDWT to recover the array and then the audio, it sounds quite the same as the primary audio. That is to say, we successfully hide an image into audio, for the listener can't notice that.

The process of adding and extracting the watermark of the DWT algorithm is shown in Algorithm 2. In the third step, notice that the length of $cD2$ and the image array are not always the same, we just replace a few elements in the front. For example, if $cD2$ = [1,2,3], image array = [4,5,6,7,8], then after the replacement, $cD2$ = [1,2,3,7,8].

Algorithm 2 DWT Watermarking Algorithm

```
1: Read the audio file and get an array of integer.
2: Use DWT to transform this int array to 3 arrays named
   $cA2$, $cD2$ and $cD1$.
3: Transform the watermark image as an integer image ar-
   ray, and replace $cD2$ with the array.
4: Use iDWT to transform the three arrays($cA2$, $cD2$, $cD1$)
   into one array.
5: Transform the array to audio.
6: The extraction process is its inverse transformation.
```

4.3 DWT-LSB

Even though the DWT process is recoverable theoretical, in actual calculation the primary audio and the recovered audio can't be all the same. That is because in the process of calculating the answer can't always be an integer. And when the computer deals with floats, some bad things happen! However, it's quite difficult for the human eyes to find the tiny difference between the 123-degree grey and 124-degree grey, so it's not a serious problem. But if we use the same way to hide a string into the audio, then the tiny difference can cause a false decryption result. In comparison, LSB can perfectly recover the hidden watermark, but it has worse robustness. Our group comes up with a new method to hide a watermark in audio which combines the merits of DWT and LSB. We named it the "DWT-LSB" method. The idea can be summarized as Algorithm 3.

4.4 Discrete Cosine Transform

Discrete Cosine Transform (DCT) is a transformation related to Fourier, similar to discrete Fourier transformation, but only uses real numbers. The idea is to convert the audio from the time domain to the frequency domain via the DCT, look for the chunks of data in the frequency domain that cause the smallest overall change, and overwrite it with our watermark. In this study, we tried to block the audio and then DCT transform all the audio blocks. In the DCT

Algorithm 3 DWT-LSB Watermarking Algorithm

```
1: Convert the image into an array of bits $img[i]$, and convert the audio into an int array $audio[i]$.
2: Use DWT to the audio int array and change the array in the following way:
3: For $i$ in range $(0, len(img_bit_array))$:
4: If the $image\_array$ of bits $img[i] = 0$, then change $audio[i \times tms]$ into a small number $smn$;
5: If the $image\_array$ of bits $img[i] = 1$, then change $audio[i \times tms]$ into a big number, $bgn$. $smn$, $bgn$ and $tms$ are three self-selectable constants: $tms$: the bigger the better, if the $audio\_array$ length is enough. In our project we set $smn = 10$, $bgn = 100$, $tms = 2$.
6: Using iDWT to recover the $audio\_array$ and change it to audio.
```

domain, watermark components of different intensities are embedded in the DCT coefficients of image blocks according to the result of block classification. According to the characteristics of the human sensory system, we choose to embed the watermark signal into the intermediate frequency coefficients of the original audio to retain better robustness and invisibility.

The schematic diagram of DCT transformation is shown in Figure 2.

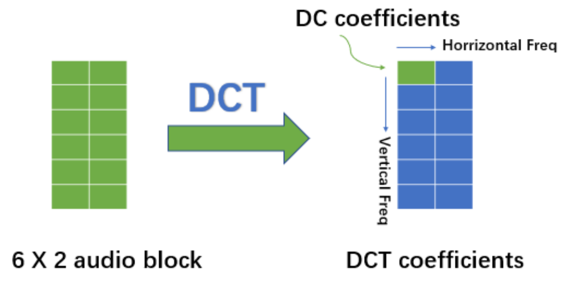


Figure 2: DCT Transformation

Formulae for DCT and inverse DCT:

$$F(u,v) = \frac{2}{N}C(u)C(v)\sum_{x=0}^{N-1}\sum_{y=0}^{N-1} f(x,y)cos[\frac{(2x+1)u\pi}{2N}]cos[\frac{(2y+1)v\pi}{2N}] \tag{3}$$

$$f(i,j) = \frac{2}{N}\sum_{u=0}^{N-1}\sum_{v=0}^{N-1} C(u)C(v)F(u,v)cos[\frac{(2x+1)u\pi}{2N}]cos[\frac{(2y+1)v\pi}{2N}] \tag{4}$$

The overall process of DCT algorithm is shown in Algorithm 4.

4.5 Robustness Analysis

The LSB algorithm embeds the watermark information into the least significant bit of the data, that is, the LSB of the data is replaced by the watermark information that needs to be added. This is precisely because the low-level data has the least impact on the overall data, which also leads to the low strength of the embedded watermark information, otherwise, it will affect the data quality of the carrier.

Algorithm 4 DCT Watermarking Algorithm

```
1: Divide the audio into non-overlapping blocks of size $row \times collum$.
2: Suppose we have $n$ such blocks.
3: Generate a set of keys called $k$. This will be used as a seed for the pseudo-random generator.
4: The size of the watermark is expressed as width $w$ times height $h$. Therefore, there are $fea = w \times h$ feature points in the watermark.
5: Check if $n \leq fea$. If not, the watermark cannot be embedded.
6: For each feature point of the watermark, repeat 7-10:
7: Compute the DCT of this block. Let it be named $dct\_block$.
8: Let $change[i] = k[i] \times watermark[h][w]$.
9: Add the $change[i]$ to the last column of the current block, $dct\_block[i, collum - 1] += change[i]$.
10: Calculate the inverse DCT of the block and save it.
11: Save watermarked audio.
```

Therefore, the algorithm is only used for fragile digital watermarking (compared with robust digital watermarking, it can not bear a lot of distortion).

By using DWT we can separate the high-frequency signals and the low-frequency signals, or the important signals and the less important signals. If we change the content in $cD1$ and $cD2$, the details of the DWT coefficients of the primary audio can't change much. When we do some bad things to the primary audio, such as add noise into it, the detail coefficients might change a little, but we replace whole integers instead of using just one bit, so when the integers change a little, the image can be shown too. Even though they can't be the same as the primary watermark, they look quite similar.

For the method combining DWT and LSB, when extracting the watermark from the encrypted audio, we compare $audio\_array[i]$ with $\frac{smn+bgn}{2}$ to decide what the current bit is. Although the calculation deviation remains, the absolute deviatation is usually smaller than 10, so the result of comparing $audio\_array[i]$ with $\frac{smn+bgn}{2}$ can hardly be changed. In this way, the watermark recovered and the primary watermark can be the same, and the audio has better robustness.

We perform DCT on the audio and obtain a DCT coefficient matrix, then embed the image information into the matrix in blocks. We chose the block positions that had the least impact on the whole data for embedding. Thus, when we attack our audio with a watermark, such as add noise or shrink it, we compute $final = IDCT(DCT)$ and right now the final contains the original floating-point values and some little noise. But when we save it as audio, the floating-point numbers are converted into integers. When the noise is small, the final integer in the audio doesn't change. But it all depends on the size of the noise, if the noise is big, it will change the integers in the audio.

In §5.3, we will evaluate the robustness of the four watermarking algorithms through actual attack cases. The final result will show the attack strength and robustness.

5. IMPLEMENTATION

5.1 Adding Watermarks

We use text as LSB watermark, image as DWT watermark, and image and audio as DCT watermark to seek diversity.

Using the LSB algorithm, we encode the audio into 0-1 matrix and replace the lowest bits to get the audio file with a watermark, as shown in the Figure 3.

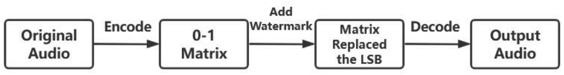


Figure 3: LSB Watermark Addition

We perform DCT/DWT on the audio to obtain a DCT/DWT coefficient matrix, embed the image information into the matrix in blocks, and then perform IDCT/IDWT on it to obtain audio with a watermark, as shown in the Figure 4.

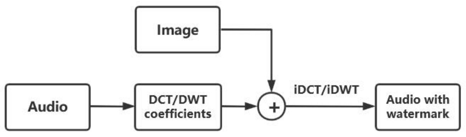


Figure 4: DCT/DWT Watermark Addition

5.2 Extracting Watermarks

The extraction process of the LSB watermark is opposite to that of adding, and the lowest bit needs to be replaced, as shown in the Figure 5.

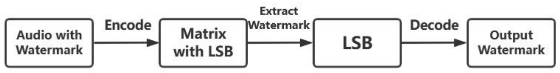


Figure 5: LSB Watermark Extraction

We perform DCT/DWT on the audio with a watermark to obtain a DCT/DWT coefficient matrix, then we decrypt the block of matrix and merge these pieces of information. After that, we got the extracted watermark, as shown in the Figure 6.

5.3 Attack & Robustness Analysis

We demonstrate the attacks of noise adding, cropping, merging, reverberation, filtering and compression, and put some samples here. If the text can be 90% restored, the image NC coefficient is greater than 0.5, and the audio correlation coefficient is greater than 0.9, we believe that it is robust under specific attacks, and vice versa. If the watermark is robust, we indicate "yes" with a "Y" in the "robustness" row. If not, we'll note the "N" in the "Robustness" line for "no".

1. **Noise Adding**
   Taking DWT as an example, the DWT original watermark is shown in Figure 7, and the DWT extracted

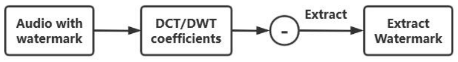


Figure 6: DCT/DWT Watermark Extraction

   watermark without attack is shown in Figure 8. Gaussian noise with a signal-to-noise ratio (SNR) of 10, 50, 100, or 150 is added to the embedded audio, and then the watermark is extracted. The results are shown in the Figure 9.

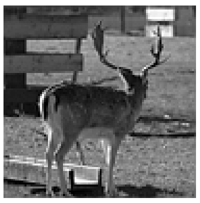


Figure 7: DWT Original Watermark

Figure 8: DWT Watermark Extraction Without Attack

   The NC coefficient of these extracted watermarks compared to the original image is 0.9906 when there is no attack, and 0.0206, 0.8554, 0.9906, and 0.9906 when the SNR is 10, 50, 100, and 150, respectively.

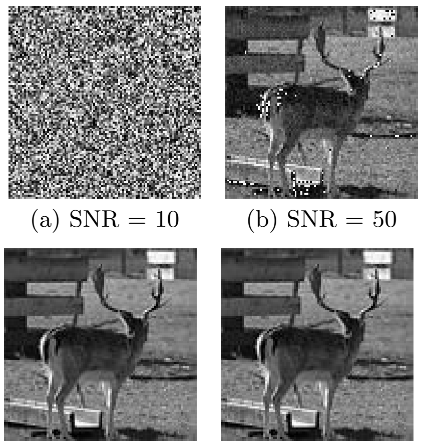
   However, the robustness of the LSB text watermark and DCT audio watermark after adding noise is very poor, the LSB text is all garbled, and the DCT audio correlation coefficient is less than 0.1. All results are shown in the Table 1.

   As we expected, the robustness of the DWT-LSB algorithm is better than that of the DWT algorithm when the SNR is large. Figure 10 shows the NC coefficients of the two methods at different SNRs.

2. **Cropping**
   Taking LSB as an example, the LSB watermark is shown in Figure 11, which can be 100% restored without attack. We crop the rear of the audio, and the text watermark restoration rate can reach more than 95%, as shown in the Figure 12. However, if the front part of the audio is cropped, garbled characters are extracted.

   For DWT, DCT and DWT-LSB, if the audio end segment is cut off, the watermark image is either not affected or cannot be generated (the array length is not enough and the program reports an error); if the audio header segment is cut, the watermark image cannot be generated. All results are shown in the Table 2.

3. **Merging**
   Merging and cropping are similar. The experimental



(a) SNR = 10 (b) SNR = 50

(c) SNR = 100 (d) SNR = 150

Figure 9: Watermark Extracted After Noise

Table 1: Performance Against Adding Noise

SNR	10	50	100	150
LSB	N	N	N	N
DWT	N	Y	Y	Y
DCT	N	N	N	N
DWT-LSB	N	Y	Y	Y

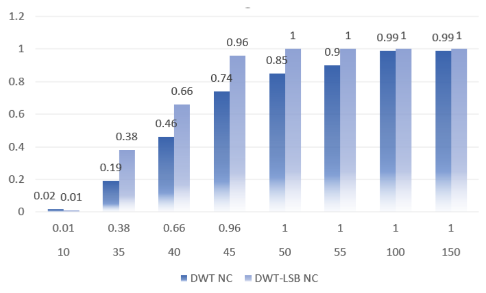
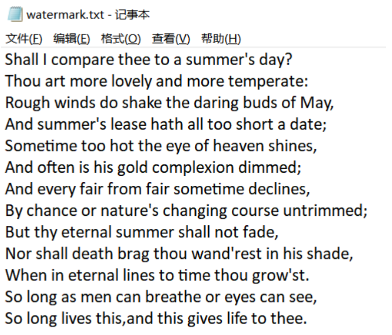
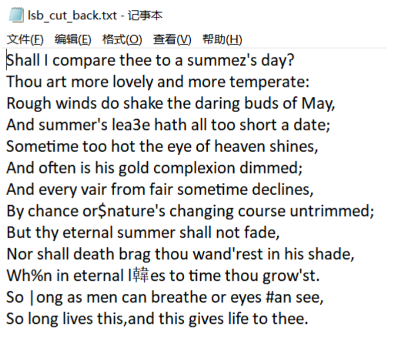


Figure 10: Comparison of DWT-LSB and DWT NC Coefficients



Shall I compare thee to a summer's day?
Thou art more lovely and more temperate:
Rough winds do shake the darling buds of May,
And summer's lease hath all too short a date;
Sometime too hot the eye of heaven shines,
And often is his gold complexion dimmed;
And every fair from fair sometime declines,
By chance or nature's changing course untrimmed;
But thy eternal summer shall not fade,
Nor shall death brag thou wand'rest in his shade,
When in eternal lines to time thou grow'st.
So long as men can breathe or eyes can see,
So long lives this,and this gives life to thee.

Figure 11: LSB Original Watermark


lsb_cut_back.txt - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

Shall I compare thee to a summez's day?
Thou art more lovely and more temperate:
Rough winds do shake the daring buds of May,
And summer's lea3e hath all too short a date;
Sometime too hot the eye of heaven shines,
And often is his gold complexion dimmed;
And every vair from fair sometime declines,
By chance or$nature's changing course untrimmed;
But thy eternal summer shall not fade,
Nor shall death brag thou wand'rest in his shade,
Wh%n in eternal l韓es to time thou grow'st.
So |ong as men can breathe or eyes #an see,
So long lives this,and this gives life to thee.

Figure 12: LSB Watermark After Tailoring

Table 2: Performance Against Cropping

Cropping	Rear	Front
LSB	Y	N
DWT	Y	N
DCT	Y	N
DWT-LSB	Y	N

results show that when a piece of audio is added to the back of the carrier audio, the robustness is good, but when a piece of audio is added to the front, the robustness is bad.

For DCT audio watermarking, the correlation coefficient before and after the attack is 0.9999. The waveforms are shown in Figure 13 and Figure 14. All results are shown in the Table 3.

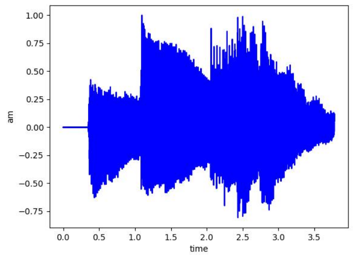


Figure 13: DCT Original Watermark

4. **Volume Changing**
Changing the volume also affects watermark extraction, and we find that DWT and DWT-LSB algorithms perform better in this attack. Figure 16 is the performance of the DWT watermark when the volume is increased or decreased by 2, 4 or 6 dB, and their NC coefficients are 0.7831, 0.6595, 0.1134, 0.9908, 0.9834, and 0.9772, respectively. Figure 17 is the performance of DWT-LSB under the same attack, and their NC coefficients are 0.9987, 0.9706, 0.8854, 1.0, 1.0, and 0.0674, respectively. DWT-LSB does not perform as well as DWT in the direction of audio volume reduction when the reduction range is large; it performs better than DWT in other audio volume adjustment directions. For space reasons, the pictures are in the appendix.

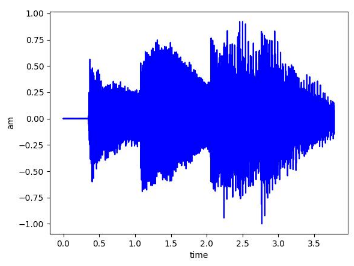


Figure 14: DCT Watermark After Merging

Table 3: Performance Against Merging

Merging	Rear	Front
LSB	Y	N
DWT	Y	N
DCT	Y	N
DWT-LSB	Y	N

LSB and DCT perform poorly on volume attacks. The overall results are shown in Table 4.

5. **Other Attacks**
We also tried filtering, reverberation, and compression attacks, all of which are not robust, and the results are shown in the Table 5.

6. A DOUBLE ENDED SCENARIO

As mentioned in the previous section, none of the four watermarking algorithms can guarantee 100% robustness, and copyright information will be destroyed under severe attacks. Therefore, we propose a watermarking and verification method in a two-terminal communication scenario, which can ensure the security of copyright information. The communication process is shown in the Figure 15. The black line indicates Alice's action and the red line indicates Bob's action.

Alice uses the SHA256 algorithm to obtain the message digest of her copyright information, and then RSA signature and AES encryption to get the final watermark, which is embedded in the audio and sent to Bob, and the initial copyright information is also sent to Bob.

After receiving the message, bob extracts the watermark from the audio containing the watermark, decrypts it, designatures it, and obtains the message digest. On the other hand, he compares the SHA256 digest of the original copyright information with the result obtained in the previous step. If it is the same, the verification passes, otherwise, it fails.

Table 4: Performance Against Changing Volume

Changing Volume(dB)	+2	+4	+6	-2	-4	-6
LSB	N	N	N	N	N	N
DWT	Y	Y	N	Y	Y	Y
DWT-LSB	Y	Y	Y	Y	Y	N
DCT	N	N	N	N	N	N

Table 5: Performance Against Other Attacks

	Filtering	Reverberation	Compression
LSB	N	N	N
DWT	N	N	N
DCT	N	N	N
DWT-LSB	N	N	N

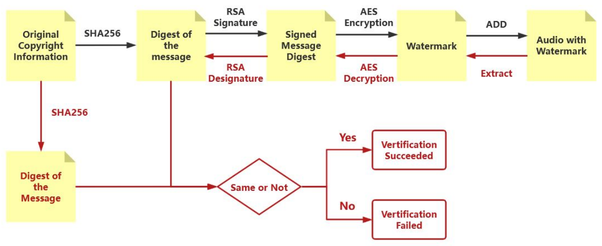


Figure 15: A Double Ended Scenario

The simultaneous use of hashing, signing and encryption ensures the data integrity and confidentiality of this protocol. In this way, the copyright information is verifiable and unforgeable.

7. CONCLUSIONS

First, we implemented three audio watermark embedding algorithms, LSB, DWT and DCT. Then, to ensure the security of watermarking, we applied them to two-terminal communication scenarios and proposed a DWT-LSB algorithm optimized based on DWT. Finally, we compare the robustness of LSB, DWT, DCT and DWT-LSB. The better robustness of DWT-LSB justifies our improved new method.

8. ACKNOWLEDGMENTS

Thank Prof. Hugh and TA Harish for their help in our project, and thank all the team members for their cooperation.

9. REFERENCES

[1] L. Boney, A. Tewfik, and K. Hamdy. Digital watermarks for audio signals. In *Proceedings of the Third IEEE International Conference on Multimedia Computing and Systems*, pages 473–480, 1996.

[2] I. Cox, J. Kilian, T. Leighton, and T. Shamoon. A secure, imperceptible yet perceptually salient, spread spectrum watermark for multimedia. In *Southcon/96 Conference Record*, pages 192–197, 1996.

[3] J. Fridrich and M. Goljan. Images with self-correcting capabilities. In *Proceedings 1999 International Conference on Image Processing (Cat. 99CH36348)*, volume 3, pages 792–796 vol.3, 1999.

[4] S. J. Lee, S. H. Jung, and IEEE. A survey of watermarking techniques applied to multimedia, 2001.

APPENDIX

A. WATERMARK UNDER CHANGING VOLUME ATTACK

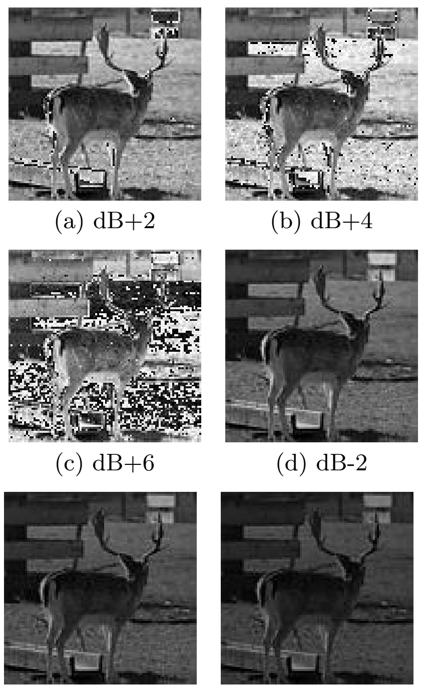

(a) dB+2 (b) dB+4
(c) dB+6 (d) dB-2
(e) dB-4 (f) dB-6

Figure 16: DWT Watermark Under Changing Volume Attack

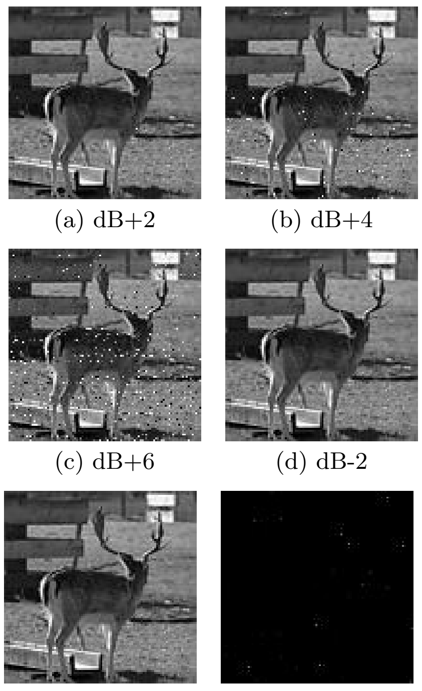

(a) dB+2 (b) dB+4
(c) dB+6 (d) dB-2
(e) dB-4 (f) dB-6

Figure 17: DWT-LSB Watermark Under Changing Volume Attack