PDFIET: PDF Malicious Indicators Extraction Technique Through Optimized Symbolic Execution

Enzhou Song, Tao Hu(✉), Peng Yi, and Wenbo Wang

Information Technology Institute, Information Engineering University, Zhengzhou 450001, China
hutaondsc@163.com

Abstract. The malicious PDF documents posed a significant threat to network security in recent years. Extracting malicious indicators from PDF documents is a critical method for subsequent analysis and detection. However, current static and dynamic extraction methods are easily interfered by evasion methods such as highly obfuscation and sandbox detection, etc. Therefore, we creatively apply optimized symbolic execution to PDF indicator extraction and propose PDFIET, a technique of PDF malicious indicators extraction consisting of three modules: code parsing, symbolic execution and indicator extraction. We design the code rewriting method to improve code coverage by enforcing branch transfers. We also use the concurrency strategy and two constraint-solving optimization methods to enhance the efficiency of symbolic execution. We use 1271 malicious samples to make several experiments. The success rate and the effectiveness of indicators is high. The code coverage and the system efficiency improve significantly after optimization. The evaluation supports the design of the approach.

Keywords: Malicious documents · JavaScript code · Indicator extraction · Optimized symbolic execution

1 Introduction

In recent years, there has been a dramatic increase in the number of cyber attacks achieved through the malicious document. It is widely used as an important vector by APT groups, resulting in phishing, vulnerability exploitation, denial of service and other serious hazards. In a variety of malicious documents, the PDF implements attacks through embedded scripts, remote links and the construction of specific vulnerabilities by the rich functionality of various reader products and the existence of security flaws in the software. Table 1 gives a number of examples of PDF exploits that have been widely threatened in recent years. As shown in the table, such attacks are highly insidious and difficult to be detected and cause serious damage such as identity extraction and arbitrary code execution. According to the statistics of F-Secure company, attacks using malicious PDF

© ICST Institute for Computer Sciences, Social Informatics and Telecommunications Engineering 2025
Published by Springer Nature Switzerland AG 2025. All Rights Reserved
H. Duan et al. (Eds.): SecureComm 2023, LNICST 568, pp. 409–425, 2025.
https://doi.org/10.1007/978-3-031-64954-7_21

Table 1. Example of pdf vulnerability exploitation

Number	Reader Product	Vulnerability Causes	Hazard
CVE-2022-28672	Foxit Reader	Improper permission control	Arbitrary code execution
CVE-2022-27787	Adobe Reader	Buffer overflow	Arbitrary code execution
CVE-2022-24934	WPS Office	Improper permission control	Arbitrary code execution
CVE-2021-44709	Adobe Reader	Buffer overflow	Arbitrary code execution
CVE-2021-21045	Adobe Reader	Improper permission control	Arbitrary code execution
CVE-2020-14425	Foxit Reader	Improper permission control	Arbitrary code execution

accounted for more than 80% in the total number of document attacks in 2020, which posed a serious threat [1–3].

Targeting on the threat posed by malicious PDF, researchers have proposed relevant analysis and detection methods in recent years. It is common to detect malicious document by analyzing the JavaScript code embedded in the document [4]. The analysis lies in extracting the hidden malicious indicators in the code as richly as possible. Currently, extraction methods are mainly divided into two categories: static and dynamic techniques. The static technique directly uses anti-obfuscation method without loading and executing the code. Combining with the document structure, it parses variables and strings of the code and other related information as extraction indicators [5]. The dynamic technique usually executes the extracted code in simulation. It uses suspicious Shellcode discovered through memory monitoring, system behavior and sequences, etc. as extraction indicators [6,7].

The static technique can better recover the original semantics of the code with higher security. It has the advantages of high detection efficiency, high speed and low overhead. However, it has limitation for highly obfuscated code and the code based on memory attacks. Some attackers can construct PDF documents with wrong formatting or take re-obfuscation techniques to interfere with the parsing and extraction. The dynamic technique makes up for the shortcomings of the static one and can fully extract information during code execution. But it is still helpless against escape tactics used by attackers such as setting specific trigger conditions, detecting execution environments and delaying execution, etc.

Based on the above analysis, we propose PDFIET, a technique of PDF malicious indicators extraction based on optimized symbolic execution. The core component of our system is a symbol execution engine designed for JavaScript code embedded in PDF documents. Through optimized symbol execution and constraint solving technique, we can infer key values used for deobfuscation. By traversing the code execution paths, we can successfully capture all possible behaviors of the JavaScript code and record relevant variables and strings.

We used two different datasets to evaluate the system. The first consists of 579 publicly collected malicious PDF samples, while the second consists of 692 privately collected samples. The datasets cover multiple typical vulnerability samples of three mainstream PDF readers over the past decade. Out of 1271

malicious samples, we successfully extract malicious indicators from 1172 samples. Furthermore, the datasets include 414 highly obfuscated malicious samples with specific triggering conditions. We successfully extract indicators from 392 of these samples.

The main contributions are summarized as follows:

1) We creatively use the symbolic execution for extracting malicious indicators from PDF documents and implement PDFIET, which can be effectively applied in malicious PDF research and detection.
2) We employ code rewriting method to the symbolic execution engine, which can address the path loss issue and maximize code execution coverage. We also design a concurrent strategy and two constraint-solving optimization methods to improve system efficiency.
3) We extract and evaluate indicators from 1271 malicious samples, which include typical vulnerabilities in three mainstream PDF readers over the past decade. We design experiments to assess the indicator extraction, indicator effectiveness, code coverage, extraction efficiency and system design.

The remaining part of the paper proceeds as follows. Section 2 introduces background knowledge and related work. Section 3 presents the design and implementation of the main system, PDFIET. Section 4 designs validation experiments and evaluates the results. Section 5 summarizes the work and provides outlook for the future research.

2 Background and Related Work

2.1 Principle of PDF Parsing JavaScript Code

The execution engine of JavaScript is an important component of PDF reading products. According to the latest ISO standard [8], the physical structure of PDF documents consists of the file header, cross-reference table, trailer, file footer and document body which contains text, images, fonts and other information. The document body is the most crucial which is composed of multiple document objects linked in a hierarchical structure. The actual parsing process of the reader is as shown in Fig. 1. It is divided into two main phases: object search and code reorganization. The reader first reads 'Root' in the trailer and 'startxref' in the file footer to find the root node which stores the document object structure. Then it iterates over objects and decodes the stream content based on encoding fields such as 'FlateDecode' and 'Type'. The decoded objects with type 'Action' and keywords '/S', '/JavaScript' and '/JS', etc. will be recorded. Finally, the reader reassembles recorded objects sequentially and executes the reassembled JavaScript code.

2.2 Related Work of PDF Malicious Indicators Extraction

Previous studies have proposed two main approaches for analyzing and extracting malicious indicators from JavaScript code embedded in PDF. The first approach involves restoring the code to its original form as much as possible through

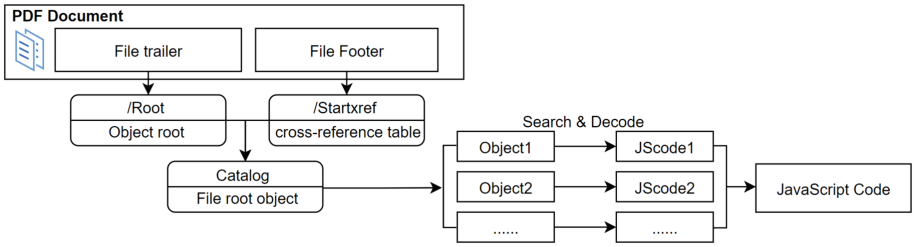

Fig. 1. Flowchart of code parsing module

techniques such as deobfuscation, without actually simulating the execution of JavaScript. It extracts information including content, keywords, document structure, etc. as indicators.

Maiorca et al. [9] extracted the frequency of keywords such as objects and JavaScript triggers as the final indicators. Wang et al. [10] focused on the code structure as indicators instead of the code content. Ndichu et al. [11] collected a total of 45 features such as keyword frequency, number of lines and characters in the code as indicators. Fraiwan et al. [12] extracted four groups of features as indicators, including URL properties, code results, code activities and code content. However, these static approaches are not effective in highly obfuscated JavaScript code and are easily influenced by re-obfuscation and other techniques. Some indicators lack obvious maliciousness and are used as feature vectors in subsequent classification models. These methods cannot detect malicious samples which utilize new features for attacks and are vulnerable to attacks from adversarial samples. Moreover, the extracted indicators cannot serve as explicit evidence for analysts to study the attack of malicious samples.

Another approach involves obtaining variable information, intermediate language and shellcode, etc. by simulating the execution of code. Laskov et al. [13] used the SpiderMonkey engine to simulate the JavaScript code and used the intermediate language as indicators. However, intermediate language can only serve as classification criteria and is difficult to represent the maliciousness of samples directly. Li et al. [14] combined with the static feature extraction techniques. They used keywords extracted from deobfuscated code and shellcode obtained from simulation execution as feature indicators. ZhuGe et al. [15] proposed MPScan, which hooked Adobe Reader and extracted suspected shellcode as feature indicators. Cova et al. [16] simulated the execution of JavaScript and extracted pre-defined features as indicators. Ma et al. [17] dynamically executed JavaScript code and design a variable name reader to read the initial and final values of variables. However, these dynamic extraction methods are not effective against documents that trigger malicious behavior under specific environments or conditions and are weak against evasion techniques.

Hu et al. [18] adopted a well-established symbolic execution approach and developed an enforcement execution engine which allowed code fragments to be executed along different paths. However, this method has limitations in extract-

ing indicators from code which utilizes specific reader API functions and may encounter issues such as overload and path explosion. Moreover, reassembling JavaScript code in PDF documents is a critical prerequisite for subsequent simulation execution.

2.3 Obfuscation and Evasion in JavaScript

Attackers can heavily obfuscate JavaScript code and design evasion techniques to interfere with the analysis methods mentioned above. As shown in Fig. 2, We construct a malicious PDF document with encrypted JavaScript code. The code exploits a vulnerability in the API function of execution policy management in Foxit Reader (CVE-2020-14425 [19]) to execute malicious scripts. We obfuscate this code and set the evasion technique that triggers only at specific times, successfully bypassing static and dynamic detection of Virustotal [20].

```
9 0 obj
<<
/S/JavaScript
/JS(varlast=newDate()["\u0067\u0065\u0074\u0054\u0069\u006d\u0065"]();if(0x1865a700c10<last)
{app["\u006f\u0070\u0065\u006e\u0063\u0050\u0044\u0046\u0057\u0065\u0062\u0050\u0061\u0067\u0065"]
("exe.clac\\\\23metsyS\\\\swodniW\\\\:C".split("").reverse().join(""));})
>>
endobj
```

Fig. 2. Sample of CVE-2020-14425

In addition, JavaScript can employ various obfuscation techniques such as variable substitution, control flow flattening and string array encoding, etc., which greatly interfere with static extraction techniques. Attackers can also employ evasion techniques such as file detection, process detection, username detection, interaction detection and delayed execution to bypass sandbox and virtual machine environments.

2.4 Symbolic Execution

Symbolic execution is a common program analysis technique which executes a program within an abstract domain of symbolic variables. It executes the code after each conditional instruction and tracks all constraints introduced during the process. For instance, the variable is initially unconstrained when JavaScript code reads an integer value related to the environment. However, when the code executes a conditional instruction $X \geq 0$, the flow will be divided into two branches, constraining variable to be positive ($X \geq 0$) or negative ($X < 0$). This analysis technique can determine the inputs which trigger specific branches of the program through constraint solving, thereby improving the code coverage. However, replacing symbolic values with actual values can result in path loss and incomplete code execution, which has been optimized by this paper.

3 PDFIET

3.1 Design Architecture

PDFIET consists of three modules: code parsing, symbolic execution and indicator extraction. The basic framework is shown in Fig. 3. The code parsing module mainly analyzes the objects of PDF and reassembles JavaScript code. The symbolic execution module is the core part of the system, which first builds the execution environment with relevant symbolic variables and performs symbolic exploration execution. Finally, it uses a constraint solver to determine the values of variables and transforms expressions from the symbolic domain to the concrete domain. The indicator extraction module mainly parses and records the return values and variable values, etc. from each execution path and generates malicious indicators.

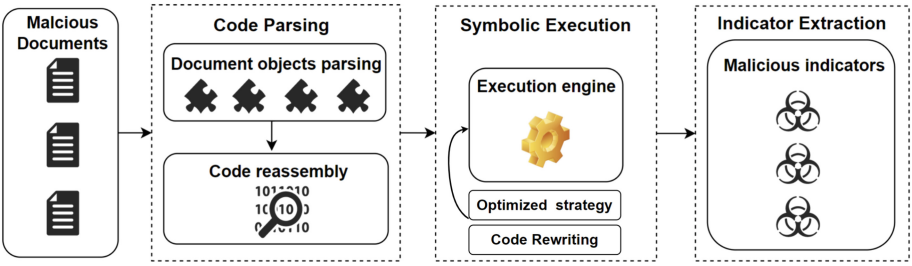

Fig. 3. Flowchart of PDFIET

3.2 Code Parsing Module

Based on the working principle described in Sect. 2.1, this module first traverses the document objects based on the document structure and performs decoding operations. Then it extracts JavaScript code based on property keywords of objects. Finally, it reassembles the extracted code according to the object order.

The first step is parsing the document structure, which mainly finds all objects in the body of the document. Since PDF readers can parse documents without standard format specifications, attackers may intentionally delete objects from the cross-reference table and keywords from the file footer to disrupt static analysis. As shown in Fig. 4, such non-compliant structures (without the cross-reference table, EOF%% symbol or endobj symbol) can still be read by PDF readers. As we can see, all objects start and end with similar symbols. Therefore, we can ignore the cross-reference table and iterate over the document to identify objects based on object forms.

Once the parser recognizes all objects, it will perform deobfuscation on each object to extract relevant information. PDF documents often use different encoding methods to hide code in stream objects, which are specified by the 'FILTER'

```
%PDF 1.4
1 0 obj <<
  /Type /Catalog
  /Pages 1 0 R
  /OpenAction <<
  /S /JavaScript
  /JS (app.alert{'There is no reference table'})
  >>
>>
endobj
Trailer <<
/Root 1 0 R >>
```

```
%PDF 1.4
1 0 obj <<
  /Type /Catalog
  /Pages 1 0 R
  /OpenAction <<
  /S /JavaScript
  /JS (app.alert{'The object format is wrong'})
  >>
>>
Trailer <<
/Root 1 0 R >>
EOF%%
```

Fig. 4. Malicious sample with irregular formatting

field. The parser first reads the encoding type and applies the corresponding decoding method to the content. It also performs additional checks on the content to restore encoding methods such as replacing ASCII characters and ultimately achieves recognition and parsing of the document objects.

The code parsing module also implements extraction and reassembly of JavaScript code after parsing the PDF document objects. According to the relevant specifications, objects containing JavaScript code are identified by keywords such as '/JS'. The code can be stored within objects or at other linked locations. The parser first determines the numbers of objects with JavaScript code and then identifies the entry points for code execution through flag fields such as '/OpenAction' and '/Names', etc. Finally, the code is reassembled based on the object sequence and linked locations.

3.3 Symbolic Execution Module

The symbolic execution module is the core module of the system, responsible for building a symbolic execution engine and solving variables through constraint solvers. The basic architecture of this module is shown in Fig. 5. It can obtain satisfying values of variables in conditional instructions, explore all valid branches of the code and obtain execution results under different paths.

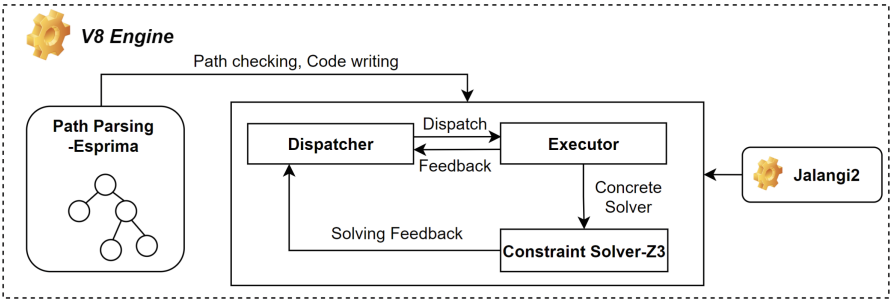

Fig. 5. Flowchart of symbolic execution module

The module first builds an execution environment of JavaScript, which is implemented based on the V8 execution engine [21]. The symbolic execution engine is implemented through the open-source tool Jalangi2 [22], which is written in JavaScript and provides callback functionality. As shown in Fig. 5, the engine mainly consists of a dispatcher, an executor and a constraint solver. The dispatcher manages the global state of symbolic exploration, aggregates relevant data and schedules test cases for symbolic execution. The executor concurrently runs multiple test cases from the program and performs symbolic tracing. The constraint solver is implemented through the open-source tool Z3 [23], which is used to solve the values of branch variables and generate new test cases returned to the dispatcher. The specific details of this execution engine are described below.

Path Parsing. To get all branches of the code, the engine first performs path parsing on the reassembled code and generates an abstract syntax tree. The path parser is implemented through the Esprima [24], which is a standard ECMAScript parser. Once all execution paths of the code are determined, the module applies code rewriting methods to improve coverage combined with the subsequent path constraints during symbolic execution.

Concrete States Generation During Execution. The executor distinguishes between concrete states and symbolic states. The executor performs a check before operation. It continues to execute the program as the source code if the value is concrete. Otherwise, it updates the path condition, negates the constraint condition, takes a symbolic execution towards a new path and returns the relevant conditions of the original path to the dispatcher.

Concurrency of Test Cases. As mentioned earlier, when the executor enters symbolic execution, the original path-related conditions are returned to the dispatcher and resubmitted to a new executor thread for execution. As a result, different test cases with their own memory and constraint condition are independent. The memory mainly includes executed statements and variable information, etc. The constraint conditions mainly include current symbolic and path constraints. The time spent on the depth-first search strategy in symbolic execution can be effectively reduced by concurrency.

Code Rewriting. Dynamic symbolic execution replaces unsolvable parts with concrete values, which can greatly reduce the inaccuracy caused by timeout of external code interactions and constraint solving. However, this approach also sacrifices the completeness of path exploration to some extent. To address this issue, the system records the executed code paths and compares with the branches obtained from path parsing after the completion of symbolic execution. The corresponding branch node of the lost path will be rewritten and executed,

thus maximizing the code coverage. For instance, The code in Fig. 6 sets a condition which is triggered at a specified time and uses XMLRequest to check the sandbox environment. The code is mainly divided into three paths by parsing module. After symbolic execution, the missed-execution path will be rewritten by code rewriting methods (For this situation, the If branch will be rename to True or False).

```
if (currentDate.getHours() >= 8 &&
  currentDate.getHours() < 20) {
  console.log("continue");

  if (SandboxCheck()) {
    setTimeout(() => {
      console.log("satisfied");
    }, 5000);
  } else {
    console.log("state unsatisfied");
  }
} else {
  console.log("state unsatisfied");
}

function SandboxCheck() {
    if (typeof window.XMLHttpRequest === 'undefined') {
        return false;
      } else {
        return true;
      }
}
```

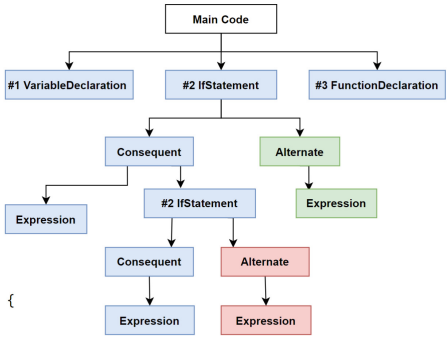

Fig. 6. Flowchart of symbolic execution module

To better measure the effects of the model, we formalize the code coverage as follows:

$$coverage_1 = \frac{record_{symbol}}{\sum_{i=1}^{n} \prod_{j=1}^{k} b_{ij}} \tag{1}$$

As the formula, $recode_{symbol}$ represents the total number of branches during symbolic execution, n denotes the number of nodes in the abstract syntax tree, k represents the number of branch nodes in the abstract syntax tree and b_{ij} represents the number of possible paths for the possibility j of the node i.

Constraint Solving Optimization. The role of the constraint solver is to generate concrete solutions and transfer symbolic values to concrete values. However, constraint solving may sometimes suffer from explosion of symbolic values. To improve the efficiency, we adopt two methods: overload optimization and adaptive optimization.

Overload Optimization. As the first formula in Fig. 7, the statement uses a symbolic expression based on 'get' to retrieve the value of a specified field name. The variable has 2^{32} concrete solutions, which can result in 2^{32} execution states after execution, leading to an overload phenomenon. Therefore, we introduce an additional symbol as an intermediate variable to represent the corresponding

operation expression such as symbolic comparison, symbolic boolean and symbolic indexing operation. For example, as the first formula, the solver identifies that the expression uses a symbolic comparison operation. It introduces a new symbol variable $Temp1$ to represent this operation. The expression is eventually simplified to $Temp1 + 80$, which has only two concrete solutions 81 and 82, greatly optimizing the efficiency of constraint solving.

Adaptive Optimization. As the second formula in Fig. 7, though the statement calls an incorrect API function, the constraint solver still attempts to concretize it. Therefore, we introduce adaptive optimization to check the validity of the formula. It filters out the formula and deletes related code paths if an invalid function is called, further improving the efficiency of constraint solving.

```
[Var 1]=Function((app.getFieldValue("test") >100) +80)  [Var 2]=Function((app.getFildeVulae("test") >100) +80)
```

Fig. 7. Samples of constraint solving optimization

When constraint solver optimization is triggered, we need to exclude invalid branches when calculating the code coverage. Therefore, we define the set ω as the collection of invalid branch nodes. The formal representation of code coverage is now as follows:

$$coverage_2 = \frac{record_{symbol}}{\sum_{i=1}^{n-len(\omega)} \prod_{j=1}^{k} b_{ij}} \tag{2}$$

3.4 Indicator Extraction Module

The indicator extraction module is mainly used to extract malicious indicators from the code. Based on the variable values obtained through symbolic execution and the concrete values obtained from constraint solving, the module generates change records for each variable. The records containing malicious indicators are finally filtered as a comprehensive indicator report by the regular matching method. The indicators we extract are mainly divided into 4 categories.

1) Network-based indicators, including IP addresses, domains, URLs, ports and network traffic patterns.
2) Host-based indicators, including file names, register keys, hashes and process names.
3) Behavioral indicators, including system commands, shellcode and executable code.
4) Metadata indicators, including author names, time and version details.

In order to record the changes of variable values in the correct order, the specific process is shown in Algorithm 1. Firstly, all variable names and definition ranges are sequentially recorded based on the syntax tree. Then, the execution ranges of each line of code are parsed and recorded. Subsequently, the mapping relationship between code and variables is established. Finally, the variable values are recorded in the correct order according to the symbol execution records.

Algorithm 1. Algorithm of indicator extraction

Input: JSCode,Record
Output: result_list([[identifier,value],...])
//Step 1: Extract variable names and definition ranges
for each $identifier \in JSCode.Tokens$ **do**
  identifier_list.append(identifier, identifier.range)
end for
//Step 2: Extract code snippets and definition ranges
for each $code \in JSCode.Syntax$ **do**
  code_list.append(code, code.range)
end for
//Step 3: Establish mapping between code and variables
for each $number \in code\_list$ **do**
  code_list.append(code, code.range)
  if identifier_list[range] in code_list[range] **then**
    code_with_id_list.append(code,identifier)
  else
    continue
  end if
end for
//Step 4: Read and insert variable values into result list
result_list=Read_and_Insert(Record,code_with_id_list)
return result_list

4 Experiment

4.1 Experiment Settings

In order to verify the performance of the system, we conduct tests from four aspects.

1) Code coverage. It focuses on the path coverage achieved during symbolic execution, which demonstrates the optimization of the code rewriting method.
2) Indicator extraction. It involves testing the ability to extract indicators from samples successfully of the system, which reflects the reliability of the implementation and system design.
3) Indicator effectiveness. It evaluates the effectiveness of metric extraction by comparing the triggering rate of YARA rules with mainstream JavaScript deobfuscation tools (JSDetox [25], JS-beautify [26], Prepack [27]).
4) Extraction efficiency. It investigates the efficiency improvement achieved by the constraint solving optimization techniques designed in this paper.

We totally collected 1271 malicious samples, including 579 samples from publicly available datasets and 692 samples from private datasets. The dataset covers multiple typical vulnerability samples from three mainstream PDF readers over the past decade. Among them, there are 414 complex malicious samples with high-level obfuscation techniques and specific triggering conditions.

The experiments were conducted on a system with the following specifications: Intel(R) Core(TM) i7-1165G7 @ 2.80GHz CPU, 16.0GB RAM, running Ubuntu 18.04 as the operating system. The software used for implementation was Python 3.6.9, Node.js 12.18.1, and Esprima 4.0. The maximum allowed testing time for each sample was 10 min.

4.2 Code Coverage

In this experiment, we investigated the code coverage of symbolic execution. As shown in Table 2, most of the samples had high code coverage without code rewriting mechanism, indicating the reliability of the symbolic execution system. The proportion of samples with code coverage between 0.9 and 1.0 increased by 8.3% with code rewriting mechanism. Since the system rewrites all unexecuted path branches except for cases due to syntax errors, the code coverage can reach 100%. As shown in Fig. 9, the average code coverage increased by 8.5%. It can be seen that after the trigger of the code rewriting mechanism, the average code coverage significantly increased after the first round of symbolic execution completed in 6.1 s, demonstrating the effectiveness of this mechanism.

Table 2. Proportion of samples with different code coverage

Code coverage	0–0.5	0.5–0.6	0.6–0.7	0.7–0.8	0.8–0.9	0.9–1.0
Sample proportion before code rewriting	0.06	0.02	0.02	0.04	0.02	0.84
Sample proportion after code rewriting	0.04	0.01	0.01	0.01	0.01	0.92

It is worth noting that the proportion of samples with code coverage between 0 and 0.5 is higher than which between 0.6 and 0.9. The code rewriting can ensure the traversal of all branches in cases where the rewritten code does not have syntax or formatting errors. However, the code coverage may significantly decrease when the rewritten code cannot be executed properly.

4.3 Indicator Extraction

In this experiment, we investigated the results and success rate of indicator extraction. Based on whether symbolic execution was triggered during the process, we classified the regular and complex malicious samples into two categories: concrete execution and symbolic execution. The extraction result is shown in Fig. 8. As the extraction result, the success rate of indicator extraction for regular samples was 96.5%, while the success rate for complex samples was 83.3%. The overall success rate for all samples was 92.2%. We counted the number of different categories and samples for malicious indicators, as shown in Table 3.

However, there were cases of failure in both actual and symbolic execution. Firstly, some API functions called in the samples were not supported by the system. Secondly, there were code snippets with syntax errors after JavaScript

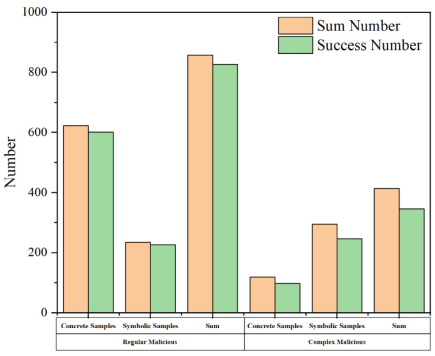

Fig. 8. Success Number of indicator extraction

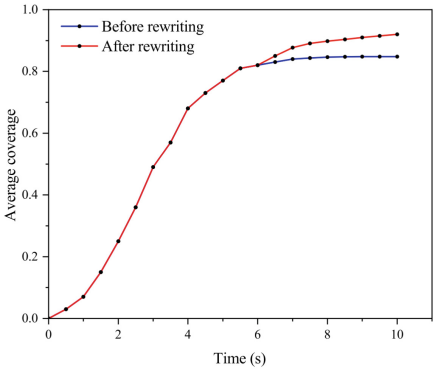

Fig. 9. Average code coverage

Table 3. Number of categories and samples for malicious indicators

Indicator Category	Network-based	Host-based	Behavioral	Metadata
Number of indicators	375	119	586	314
Number of samples	336	117	421	298

code recombination. Thirdly, some entry points of the code with heavy obfuscation were not successfully located. Compared to regular malicious samples, the success rate of indicator extraction for complex malicious samples decreased significantly, indicating that the various evasion techniques employed by attackers posed interference on the indicator extraction process.

4.4 Indicator Effectiveness

The experiment aims to investigate and demonstrate the effectiveness of the system in extracting malicious indicators and conducts comparative experiments to study the role of extracted indicators in PDF detection. We used self-developed YARA rules as the basis for maliciousness detection, which mainly match sensitive filenames, URL domains, Shell commands and execution functions (such as downLoad, EXEC, Fopen, Fwrite, http, process, etc.). In this experiment, we selected three mainstream JavaScript deobfuscation tools (JSDetox, JS-beautify, Prepack) for comparison.

First, we utilized selected tools to deobfuscate the reconstructed JavaScript code and then extracted variables and string information as indicators, which along with the indicators extracted by PDFIET were separately input into YARA rules. The successful triggering of the rules demonstrated that the detection of malicious documents can be effectively achieved based on the indicators. The detection probabilities and quantities of our system and other tools are

shown in Table 4. The total number of malicious script attack samples was 872, with a detection rate of 90.9%. The total number of Shellcode attack samples was 399, with a detection rate of 93.3%. The overall effectiveness of indicator extraction was 91.7%.

Table 4. Detection proportion and number of different tools

Tool	Malicious type and detection proportion			
	Proportion(Script)	Number(Script)	Proportion(Shellcode)	Number(Shellcode)
JSDetox	61.7%	538	17.9%	71
JS-beautify	54.8%	478	17.1%	68
Prepack	69.2%	603	18.1%	72
PDFIET	90.9%	793	93.3%	372

As shown above, mainstream deobfuscation tools have a certain effectiveness in detecting malicious script attacks, indicating that the deobfuscation techniques used to restore code semantics, structure and critical variables are effective in extracting malicious indicators. However, there are still significant false negatives. Since some malicious codes employ techniques such as encoding functions to obfuscate and hide, it difficult to make extraction through static analysis and code reconstruction. In addition, mainstream deobfuscation tools have low detection rates for Shellcode attacks similarly. As most memory attackers hide and encode their executable code, it challenging to extract indicators using common deobfuscation techniques.

Due to the dynamic indicator extraction approach based on symbolic execution employed in our system, the detection effectiveness for malicious PDF has been significantly improved compared to mainstream tools. The system performs better in extracting indicators for Shellcode attacks compared to malicious script. Because JavaScript code used in memory attacks tends to have simple functionalities such as variable allocation and stack manipulation. It has less code obfuscation types, making it easy to extract Shellcode through dynamic tracking of variables. On the other hand, malicious script attacks exhibit diverse and flexible evasion techniques, which to some extent reduce the success rate of indicator extraction.

4.5 Extraction Efficiency

The experiment mainly explores the extraction efficiency and the improvement effect of constraint solving optimization. The average and median time spent on indicator extraction by each module of the system are shown in Table 4. All samples were resolved within the 10-minute time limit. The system took 7.79 s on average and 7.51 s on median, which has a high efficiency of extraction (Table 5).

As described in Sect. 3.3, constraint solving optimization is divided into two parts: overload and adaptive optimization. We explored two types of optimization methods separately. The experiments are tested on 1271 malicious samples,

Table 5. System time spent

Extraction phase	Average time spent(s)	Median time spent(s)
Code parsing	0.82	0.77
Symbolic execution	7.43	7.18
Indicator extraction	0.079	0.077
Total time	7.79	7.51

among which 529 malicious samples trigger symbolic execution. We recorded the number of symbolic execution branches and the average time of extraction for such samples before and after the constraint solving optimization. We classified the samples into four categories: without optimization, overload-only optimization, adaptive-only optimization and overload-adaptive optimization. The results are shown in Fig. 10.

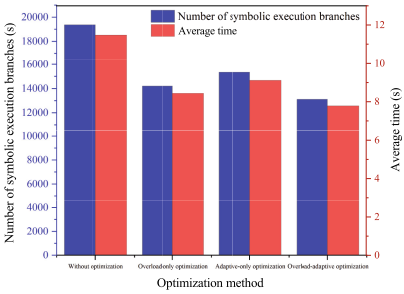

Fig. 10. Number of branches and average time

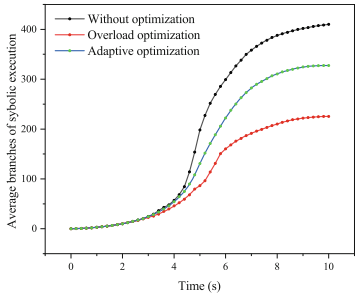

Fig. 11. Change of branches

As shown in figure, the number of symbolic execution branches is reduced from 19377 to 13116, and the average time spent for indicator extraction is reduced from 11.47 s to 7.79 s after introducing the optimization method, which improves the system efficiency by 32.3%. Both overload and adaptive optimization can effectively reduce the number of branches and improve the efficiency of indicator extraction. It is worth noting that the overload optimization has a higher effect compared with the adaptive one. During the test, a total of 21 samples triggered overload optimization and 13 samples triggered adaptive optimization.

We recorded the average branch number of the optimization samples, as shown in Fig. 11. The samples caused branch explosion without optimization. Two optimization methods can better suppress the phenomenon and reduce the average branch number to 55% and 79.9% of the original, effectively preventing the system overload.

5 Conclusion

The paper innovatively applies symbolic execution technology to the field of PDF malicious document analysis. We implement PDFIET, a malicious indicator extraction technology for PDF documents and design three modules: code parsing, symbolic execution and indicator extraction. The core of the system is the symbolic execution engine. The performance of the engine is optimized through concurrency policy and constraint solving optimization. We design experiments on the success rate of indicator extraction, code coverage, indicator effectiveness and extraction efficiency. The experimental results show that the system can efficiently extract indicators from PDF documents with high success rate, effectiveness and efficiency. It can effectively counter the common obfuscation and escape tactics of attackers. It provides a good basis for subsequent research on PDF malicious code and maliciousness detection.

There are also some limitations of the system. First, weakness of API functions. The symbolic execution will be affected by some API functions only supported by the reader product. Such functions result in failure of execution. Second, syntax errors. The code path with syntax will exit abnormally, which significantly affect the symbolic execution. In the future work, combined with the open source of related reader products, we can enrich and expand more API functions. We can also try to remove wrong code sentence to improve the execution success rate.

References

1. Lei, J., Yi, P., Chen, X., Wang, L., Mao, M.: PDF document detection model based on system calls and data provenance. J. Comput. Appl. **42**(12), 3831–3840 (2022)
2. Lu, X., Wang, F., Jiang, C., Lio, P.: A universal malicious documents static detection framework based on feature generalization. Appl. Sci. **11**(24), 12134 (2021)
3. Nissim, N., Cohen, A., Glezer, C., Elovici, Y.: Detection of malicious PDF files and directions for enhancements: a state-of-the art survey. Comput. Secur. **48**, 246–266 (2015)
4. Yu, M., Jiang, J.G., Li, G., Liu, C., Huang, W.Q., Song, N.: A survey of research on malicious document detection. J. Cyber Secur. **6**(3), 54–76 (2021)
5. Wang Y.: The de-obfuscation method in the static detection of malicious PDF documents. In: 2021 7th Annual International Conference on Network and Information Systems for Computers, ICNISC, pp. 44–47. Guiyang, China (2021). https://doi.org/10.1109/ICNISC54316.2021.00016
6. Gao, X., Yu, M., Jiang, J.G., Qiu, X.L., Liu, C.: A combined malicious documents detecting method based on emulators. Appl. Mech. Mater. **602–605**, 1707–1712 (2014)
7. Alazab, A., Khraisat, A., Alazab, M., Singh, S.: Detection of obfuscated malicious javascript code. Future Internet **14**(8), 217–231 (2022)
8. ISO32000-1:2020. https://www.pdfa.org/resource/iso-32000-pdf/
9. Maiorca, D., Giacinto, G., Corona, I.: A pattern recognition system for malicious PDF files detection. In: Perner, P. (ed.) MLDM 2012. LNCS (LNAI), vol. 7376, pp. 510–524. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31537-4_40

10. Wang, T., Mou, Z.H., Zhang, Z.H.: Detecting obfuscated malicious Javascript code based on function call information. Comput. Simul. **38**(2), 432–437 (2021)
11. Ndichu, S., Kim, S., Ozawa, S.: Deobfuscation, unpacking, and decoding of obfuscated malicious JavaScript for machine learning models detection performance improvement. CAAI Trans. Intell. Technol. **5**(3), 184–192 (2020)
12. Fraiwan, M., Al-Salman, R., Khasawneh, N., Conrad, S.: Analysis and identification of malicious Javascript code. Inf. Secur. J. Global Perspect. **21**(1), 1–11 (2012)
13. Laskov P, şrndić N.: Static detection of malicious Javascript-bearing PDF documents. In: Proceedings of the 27th Annual Computer Security Applications Conference, ACSAC, vol. 2011, pp. 373–382. Orlando, Florida, USA (2011). https://doi.org/10.1145/2076732.2076785
14. Li, M., Zhou, Y., Yu, M., Liu, C.: Combining static and dynamic analysis for the detection of malicious JavaScript-bearing PDF documents. In: Proceedings of the 2016 International Conference on Computer Science, Technology and Application, CSTA, pp. 475–482. Changsha, China (2017). https://doi.org/10.1142/9789813200449_0059
15. Lu, X., Zhuge, J.W., Wang, R.Y., Cao, Y., Chen, Y.: De-obfuscation and detection of malicious PDF files with high accuracy. In: 2013 46th Hawaii International Conference on System Sciences, pp. 4890–4899. IEEE, Wailea, HI, USA (2013). https://doi.org/10.1109/HICSS.2013.166
16. Cova, M., Kruegel, C., Vigna, G.: Detection and analysis of drive-by-download attacks and malicious javascript code. In: Proceedings of the 19th International Conference on World Wide Web, pp. 281–290. ACM, Raleigh, North Carolina, USA (2010). https://doi.org/10.1145/1772690.1772720
17. Ma, H.L., Wang, W., Han, Z.: Detecting and de-obfuscation obfuscated malicious JavaScript code. Chin. J. Comput. **40**(7), 1699–1713 (2020). https://doi.org/10.11897/SP.J.1016.2017.01699 https://doi.org/10.11897/SP.J.1016.2017.01699
18. Hu, X., Cheng, Y., Duan, Y., Henderson, A., Yin, H.: JSForce: a forced execution engine for malicious Javascript detection. In: Security and Privacy in Communication Networks, LNICST, vol. 238, pp. 704–720. Niagara Falls, ON, Canada (2017). https://doi.org/10.1007/978-3-319-78813-5_37
19. CVE-2020-14425. https://cve.mitre.org/cgi-bin/cvename.cgi?name=2020-14425
20. Virustotal. https://www.virustotal.com/gui/home/upload
21. Nodejs. https://github.com/nodejs/Release
22. Jalang2. https://github.com/Samsung/jalangi2
23. Z3. https://github.com/Z3Prover/z3
24. Esprima. https://github.com/jquery/esprima
25. Jsdetox. https://github.com/svent/jsdetox
26. Js-beaturify. https://github.com/beautify-web/js-beautify
27. Prepack. https://github.com/facebookarchive/prepack