IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received September 16, 2019, accepted November 7, 2019, date of publication November 14, 2019,
date of current version December 26, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2953511

Leveraging WebAssembly for Numerical JavaScript Code Virtualization

SHUAI WANG[1,2], GUIXIN YE[1,2], MENG LI[1,2], LU YUAN[1], ZHANYONG TANG[1,2], HUANTING WANG[1,2], WEI WANG[1,2], FUWEI WANG[1,2], JIE REN[3], DINGYI FANG[1,2], AND ZHENG WANG[4,5]

[1]School of Computer Science and Technology, Northwest University, Xi'an 710127, China
[2]Shaanxi International Joint Research Centre for the Battery-Free Internet of Things, Xi'an 710127, China
[3]School of Computer Science, Shaanxi Normal University, Xi'an 710062, China
[4]School of Computing, Lancaster University, Lancaster LA1 4YW, U.K.
[5]School of Computer Science and Technology, Xi'an University of Posts and Telecommunications, Xi'an 710121, China

Corresponding author: Fuwei Wang (wfw@nwu.edu.cn)

This work was supported in part by the NSFC under Grant 61672427 and Grant 61672428, in part by the International Cooperation Project of Shaanxi Province under Grant 2019KW-009 and Grant 2017KW-008, in part by the Key Research and Development Project of Shaanxi Province under Grant 2017GY-191, in part by the Shaanxi Science and Technology Innovation Team Support Project under Grant 2018TD-O26, and in part by the Ant Financial through the Ant Financial Science Funds for Security Research.

ABSTRACT Code obfuscation built upon code virtualization technology is one of the viable means for protecting sensitive algorithms and data against code reverse engineering attacks. Code virtualization has been successfully applied to programming languages like C, C++, and Java. However, it remains an outstanding challenge to apply this promising technique to JavaScript, a popular web programming language. This is primarily due to the open visibility of JavaScript code and the expensive runtime overhead associated with code virtualization. This paper presents `JSPro`, a novel code virtualization system for JavaScript. `JSPro` is the first JavaScript code obfuscation tool that builds upon the emerging WebAssembly language standard. It is designed to provide more secure code protection but without incurring a significant runtime penalty, explicitly targeting numerical JavaScript kernels. We achieve this by first automatically translating the target JavaScript code into WebAssembly and then performing code obfuscation on the compiled WebAssembly binary. Our design has two advantages over existing solutions: (1) it increases the code reverse entering complexity by implementing code obfuscation at a lower binary level and (2) it significantly reduces the performance impact of code virtualization over the native JavaScript code by using the performance-tuned WebAssembly language. We evaluate `JSPro` on a set of numerical JavaScript algorithms widely used in many applications. To test the performance, we apply `JSPro` to four mainstream web browsers running on three distinct mobile devices. Compared to state-of-the-art JavaScript obfuscation tools, `JSPro` not only provides stronger protection but also reduces the runtime overhead by at least 15% (up to 38.2%) and the code size by 28.2% on average.

INDEX TERMS Code obfuscation, javascript, webassembly, security, performance.

I. INTRODUCTION

Unauthorized reverse engineering and modification of code is a major concern for software vendors [1]–[3]. Such activities are often associated with malicious behaviors such as cheating, unauthorized use of software, or bypassing security and authentication mechanisms.

By making sensitive code hard to be traced and analyzed, code obfuscation offers a solution for unauthorized code reverse engineering [4], [5]. In recent years, code virtualization [6], [7] is emerging as a promising method for implementing code obfuscation. This technique translates the native instructions into bespoke virtual instructions to increase the strength of code obfuscation [8], [9]. The idea behinds code virtualization is that by forcing the attacker to work on a new, unfamiliar instruction set, one can significantly increase the difficulties of launching code reverse engineering attacks [4].

Despite code virtualization techniques have been successfully applied to native program binary [10] and program

The associate editor coordinating the review of this manuscript and approving it for publication was Chi-Yuan Chen.

languages like C [11], C++ [12] and Java [13], it remains unclear how this promising technology can be used for JavaScript, a popular programming language for web-based applications. The challenge mostly comes from two aspects. First, JavaScript is an interpreted language, and applications written in JavaScripts are often shipped in the form of plain source code. As a result, prior works on JavaScript code obfuscations [14] typically perform at the source code level, and the obfuscated program will be released again as JavaScript source code. However, the rich semantics at the source code level provide much information needed for reconstructing a simpler yet semantically equivalent code version where reserve engineering can be easily performed [15]. This drawback limits the strength of protection a code obfuscation can provide for JavaScript. Secondly, because JavaScript code is interpreted and just-in-time compiled for execution, the additional code associated with code virtualization can lead to significantly runtime overhead [16]. Such overhead often leads to poor user experience for interactive and computation-intensive applications. This prevents code virtualization to be adopted for JavaScript although it just needs much attention for code protection as other programming languages.

This work aims to unlock the potential of code virtualization on JavaScript, and it aims to do so without paying the cost of significant runtime overhead. Our work is enabled by the recently proposed WebAssembly standard [17]. WebAssembly is a new type of code in binary instruction format, and it is widely supported by and built into major web browsers including Firefox, Chrome, Safari and Microsoft Edge. Thanks to the ahead-of-time compilation feature [18], WebAssembly can run at nearly native speed by taking advantages of hardware features that are commonly available on modern computing devices.

The performance-tuned and ahead-of-time compiled WebAssembly brings in two potential benefits over prior methods that solely perform code obfuscation at the JavaScript source code level. Firstly, WebAssembly is supported by major web browser vendors. This means that we can translate the target JavaScript code into WebAssembly to perform code obfuscation at the compiled binary. As code obfuscation is done at a lower level instead of at the source code, this increases the difficulty for code analysis [19]. Secondly, the heavily optimized WebAssembly code can amortize the overhead brought by code virtualization, resulting in a code version that has only modest runtime overhead compared to the vanilla JavaScript implementation. The saving in the running time is essential for removing the barriers of applying code virtualization on JavaScript code.

However, translating this high-level idea into a practical system is non-trivial because of two specific challenges. Firstly, JavaScript uses JIT compilation to compile the source code while WebAssembly uses AOT pre-compilation. Since they work at different layers of the runtime software stack, we need to have a way to automatically translate the JavaScript code into WebAssembly and to seemingly switch between the two different execution modes during runtime. Secondly, JavaScript is a weakly and dynamically typed language, i.e., the type of data and variables can change dynamically during execution time [20]. By contrast, WebAssembly is a strongly typed language and requires the data type to be known at compile time. As a result, any code translation scheme from JavaScript to WebAssembly must be to ensure the type inferring process is automatic and precise.

This paper presents `JSPro`, a novel code virtualization system for JavaScript code. `JSPro` is the first JavaScript code obfuscation scheme built upon the recently proposed WebAssembly standard. It is designed to specifically target the computation-intensive numerical JavaScript kernels, which are commonly available in many JavaScript applications including image processing, cryptography and gaming. When implementing `JSPro`, we proposed a set of new methods, analysis and algorithms to overcome the aforementioned challenges for using WebAssembly to protect JavaScript code. Specifically, We divide the JavaScript code into computational code and DOM code. The computational code is virtualized and stored in a WebAssembly file. When running the protected JavaScript code, the WebAssembly file is first loaded to provide API functions. Then the DOM code invokes the WebAssembly API to execute the computational code. It's to note that we don't virtualize the computational JavaScript code whose parameters contain object type. The reason is that this kind of code virtualization will increase the time cost dramatically. Furthermore, we probably need to use more policies to ensure the translation correctness between object type operation with no object type operation. When encountering this case, we apply inline assembly to translate the code to WebAssembly.

We evaluate `JSPro` by applying it to 10 widely used numerical JavaScript kernels on four mainstream web browsers including Chrome, Firefox, Safari and Microsoft Edge. Our testing platforms including three distinct mobile devices where response time is a key constraint. Compared to state-of-the-art commercial JavaScript obfuscation tools, `JSPro` provides stronger code protection, and it achieves this by delivering at least 15% (up to 38.2%) faster running time with an average 28.2% reduction in the code size.

The key contribution of this paper is a novel code virtualization system for JavaScript built on WebAssembly. Our work demonstrates that by carefully exploiting WebAssembly, one can build a code obfuscation scheme that provides strong code protection for JavaScript without introducing significant runtime overhead. We hope our work can encourage further work in protecting JavaScript applications against code reverse engineering.

II. BACKGROUND

A. JAVASCRIPT SECURITY AND PERFORMANCE

JavaScript and web applications often suffer from injection-based attack [21]. At the same time, there is another issue, the accessibility of the application specific code of the applications. To prevent piracy or at least make it more

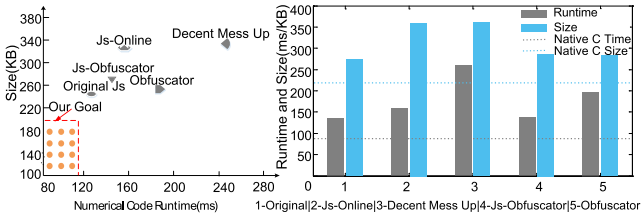


FIGURE 1. We show the protection performance results of four types of obfuscation methods focus on code obfuscation. The optimum result is that the protected code run less time than native C, and the size is smaller than the source code. The rectangular area in the figure is our design goal.

difficult, we often use encryption and obfuscation tools to keep JavaScript code safe [22]. Code encryption usually encrypts the code in the form of a string and then executing through the eval function. The main idea of code obfuscation is to convert the source code into a functionally equivalent form, but it is difficult to be understood. According to the purpose, we can category the obfuscation into the following forms, 1. Layout obfuscation, 2. Name obfuscation 3. Data obfuscation 4. Control flow obfuscation [23]. As an interpretative language, all these obfuscation methods will seriously affect the JS running performance. As shown in Figure 1, the traditional obfuscation algorithm will increase the runtime and size expansion of the program.

B. WEBASSEMBLY

In modern browsers, like `Chrome`, `Firefox`, `Edge`, or `Safari`, the code is interpreted and executed by the JavaScript engine, which only runs JavaScript. Unfortunately, JavaScript is not ideal for every task we want to perform. WebAssembly defines a text format that will be rendered when developers view the source of a WebAssembly module with any development tool. It is a binary instruction format for a stack-based virtual machine. WASM is designed as a portable target for compilation of high-level languages like `C/C++/Rust` so that it could support the deployment of client and server applications on the web [17]. As mentioned, obfuscation and encryption of JavaScript source code are easy to be cracked, so we use WebAssembly to rewrite instruction for numerical JavaScript protection. Compared to JavaScript source code, WebAssembly code is more difficult in semantic understanding. Furthermore, WebAssembly has better performance in running time, so we can conclude that JS protection with WebAssembly can speed up code execution while reducing the size of the protected code.

III. THE THREAT MODEL

Our work aims to protect JavaScript code against code reverse engineering attacks. In this work, we assume that the adversary can launch the attack from a malicious host and has access to sophisticated tools for tracking and analyzing the program behaviors. This is a reasonable assumption as many of the JavaScript applications run on a client device (e.g., a web browser). Our threat model assumes the attacker has access to the target JavaScript code and can locate, copy and

```
1  inline void FFT(Complex a[]) {
2    //pos[i] reverse position of
     representative
3    for(...)
4    if(i < pos[i])
5      swap(a[i], a[pos[i]]);
6    //len is representing the highest
     order of polynomials
7    for(...) {
8      //The first level i is where the
     enumeration is merged
9      Complex wm(cos(2.0 * pi / i), sin
     (2.0 * pi / i));
10     //The second level j is the
     enumeration merge interval
11     for(int j = 0; j < len; j += i) {
12       Complex w(1, 0);
13       //The third level k is the
     subscript in the enumeration interval
14       for(int k = j; k < j + mid; k++,
      w = w * wm) {
15         Complex l = a[k], r = w * a[k
      + mid];
16         a[k] = l + r;
17         a[k + mid] = l * r;
18       }
19     }
20   }
21 }
```

FIGURE 2. The Fast Fourier Transform (FFT) function is applied to a random data set, and which is used to evaluate the performance and size increasement.

modify the code. We also assume the attacker can intervene in the execution of the JavaScript host, e.g., a web browser. This can be achieved by running the JavaScript code in a modified open-source web browser. We note that to launch a code reverse engineering attack in our scenario will require the attacker to understand the logic of the target application through static or dynamic code analysis.

Protecting JavaScript code from such an environment is highly challenging as the attacker has access to the code and an unprotected execution environment. This work aims to increase the difficulties of tracing and understanding the target program's logic.

IV. MOTIVATING EXAMPLES

Generally, the C program not only runs faster than JavaScript but also has a smaller size. It can be illustrated in Figure 1 that presents the comparison results. We take the C code in Figure 2 as an example. The gray dot line in Figure 1 shows the run time, and the blue dot line shows the size of this example. As can be seen, the cost values are all lower than that of JavaScript execution shown as the histograms. The obfuscated code size are almost identical is because the garbage instructions dominate the obfuscated code size. Since garbage code does not get executed, it does not affect the running time; but due to the difference in security strength, we observe different running time between 2-Js-Online and 3-Decent-Mess. The combination of these two figures illustrates that exist obfuscation methods applied to JavaScript protection

```
1  var _0xd210 = ["\x31\x32\x33\x34\x35\x36\x37\x38", "\x70\x61\x72\x73\x65",
   "\x55\x74\x66\2 , x38" , "\x65\x6E\
2  x63", "\x45\x43\x42", "\x6D\x6F\x64\x65", "\x50\x6B\x63\x73\x37", "\x70\x61\x64",
3  "\x65\x6E\x63\x72\x79\x70\x74", "\x41\x45\x53", "\x64\x65\x63\x72\x79\x70\x74",
   "\x75\x73\ , x65\x72\x61\x67\
4  x65\x6E\x74"];
5  function ox00(_0x62afx3, _0x62afx4) {var encrypt =
   CryptoJS[_0xd210[9]][_0xd210[8]](_0x62afx4,
6  CryptoJS[_0xd210[3]][_0xd210[2]][_0xd210[1]](_0xd210[0]), {mode:
7  CryptoJS[_0xd210[5]][_0xd210[4]],padding: CryptoJS[_0xd210[7]][_0xd210[6]]
8  }).toString(); return encrypt}
9  function a(a) {
10 var d = !![];var e = 0x0;
11 for (; d;) {
12   switch (e) {
13     case 0x0:
14       var f, g, h;h = (g = b, f = g & 0xFFFF);
15       var i, j, k;k = (j = a, i = j & 0xFFFF);
16       var l, m, n;e = 0x1;
17       break;
18     case 0x2:
19       var G, H, I;I = (H = b, G = H >> 0x10);
20       var J, K, L;L = (K = a, J = K >> 0x10);
21       e = 0x3;break;return b;}}}
22 function oasd(_0x62afx3, _0x65) {
23 var _0x62afx6 = CryptoJS[_0xd210[9]][_0xd210[10]](encrypt,
24 CryptoJS[_0xd210[3]][_0xd210[2]][_0xd210[1]](_0xd210[0]),
25 mode: CryptoJS[_0xd210[5]][_0xd210[4]],padding: CryptoJS[_0xd210[7]][_0xd210[6]]
26 }).toString(CryptoJS[_0xd210[3]].Utf8);
27 return _0x62afx6
28 }
29 function oxad(_0x62afx4) {return 0x00(_0xd210[0], _0x62afx4)}
30 function ox001(_0x62afx9) {
31 var _0x62afx4 = _0xd210[11];
32 if (_0x62afx4 == oasd(_0xd210[0],  0xad)) {
33    return true
34  }
35 }
```

FIGURE 3. We give an example of a de-obfuscated code snippet. It shows that the adversary could crack classic JavaScript obfuscation method. After careful analysis and review of obfuscation data, it appears that the Name Obfuscation method is used from lines 1 to 4, the Control Flow Flattening method and Junk code method are used from lines 9 to 21, the String Transfer and Split method are used from lines 23 to 26. Finally, we can find that _0xd210[0] is the encryption key.

would actively expand the code volume and increase running time consumption.

Take another example in Figure 3, and it shows that the obfuscation code of the file AES.js have concise parameters and variable names, which is difficult to understand. However, after reconstructing the name, as shown in Figure 3, we can easily distinguish the position of the encryption key. Thanks for the code semantics, even for larger, more complex programs, we also can recover sensitive code elements within an acceptable timeframe for those JavaScript source level obfuscation. These obfuscation mechanisms that are running on JavaScript source code level are ineffective and easily evaded.

The two limitations posed by de-obfuscation adversaries motivate us to revisit the research of protection using a new method. We also expect that this method would race the run and reduce the code size. Furthermore, it is best to offer non-source code level protection. To satisfy the above requirements, we may then set our sights on the WebAssembly.

V. SYSTEM OVERVIEW

Our virtualization-based approach, as depicted in Figure 4, is entirely automated. As we know, WebAssembly does not currently support manipulation of DOM objects in JavaScript. JSPro separates the code of all non-calculation classes

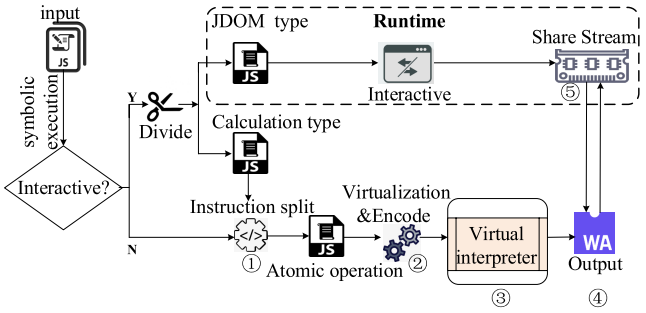


FIGURE 4. The overview of JSPro system framework. It shows that a given JavaScript is firstly divided into JDOM type and the calculation type. Then JSPro aims to translate the calculation type into the virtualization code. JSpro draw out the interactive DOM and the virtual interpreter interactive each other using share stream.

named JDOM in target JS code, allowing the separated JDOM operations and VM interpreter interactions to be implemented in shared memory so that our approach can reduce the number of interaction. For a given input JavaScript, the symbolic execution is used to extract the JDOM interactions. In this way, a pure numerical application could be easily virtualized. Further, we design virtual instructions as non-separable atomic operations (similar to assembly instructions), in order to enable virtual instructions to simulate JavaScript operations, we will split JavaScript code. After these operations are encoded and virtualized, the interpreter could start to perform instructions. However, if the input.js contains several JDOM operations, the numerical code still needs to interact with the shared stream.

VI. IMPLEMENTATION DETAILS

In this paper, we implement the virtualization process by stack virtual machine [24]. If we utilize VM to simulate the normal execution of the program, it is necessary to build the structure of the stack and register in the WebAssembly environment. Similar to other virtualization-based code protection methods, the interpreter uses a loop statement to implement the scheduling process. As described above, according to the function of input JavaScript, the code is divided into the JDOM type and calculation type. JDOM type contains JavaScript specific functions such as string processing, attributes, and so on. Calculation type includes simple numerical operations. Our system JSPro does not virtualize JDOM type code using WebAssembly. As Figure 4 shows the two parts of the code interact by sharing memory.

A. SPLIT, VIRTUALIZE AND ENCODE INSTRUCTIONS

Typical code virtualization constantly utilizes native binary instructions, which have basic operational characteristics. Different from common compiled programs, JavaScript applications are textual code with syntax attributes. Due to JavaScript does not hold the atomic properties of native operation instructions, it is difficult to be virtualized comparing to classic binary code. So how to virtualize the JavaScript code? The solution is to realize the special split conversion of the

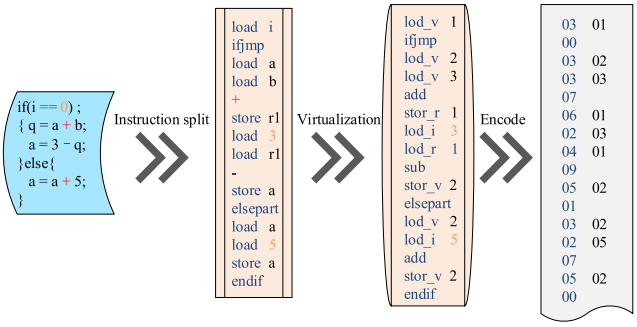


FIGURE 5. We take an example of the instruction segmentation and the process of virtualization. Due to the text attribute of JavaScript, the program state should be split firstly and then construct the instruction, which is different from C/C++ type.

target code, destroy the readable syntax attributes, extract the intermediate code with the characteristics of atomic operation, and then design the corresponding virtual instructions and processing functions. Finally, the virtualization operation is accomplished through instruction encoding. As shown in Figure 5, we will specify the virtualization process of JavaScript code with a simple example.

1) SPLIT THE INSTRUCTIONS

As shown in Figure 4 Step ①, we show a simple example in Figure 5 to illustrate why we need to split JavaScript. The expression "q=a+b" is a simple text, rather than adding semantics to JavaScript. So this expression first needs to be altered to an intermediate code sequence, which can be restored by a set of atomic operations. To achieve this goal, we extract abstract syntax trees from the input JavaScript code and split each statement to realize the separation of operands and operators. Note that in program loop statements and condition statements, we need to insert branch tags to indicate which branch to execute or whether to end the loop. After that, we finally got the atomic instructions (IR) set and complete instructions split.

2) VIRTUALIZE THE INSTRUCTIONS AND DESIGN THE HANDLES

To accomplish the virtualization of IR, JSPro offers three kinds of virtual instructions, Data transfer instruction, Control transfer instruction and Operation statement instruction. As shown in Figure 6, data transfer instructions are designed to transfer variable value to register or VarList which is contained in VMcontext. As Control transfer instruction is designed for replacing branch tags ("ifjmp","elsepart" and "endif" in Figure 5) which is inserted into IR.Operation statement is designed for mapping symbol of operation.For example, we use "add" to express "+" in Figure 5. Handler is a component for restoring semantics. We design handler corresponding to the virtual instruction. For example, for the virtual instruction "lod_r" in Figure 5, we design handler to load variable value from register. For the virtual instruction "lod_v", we design handler to load variable value from VarList.

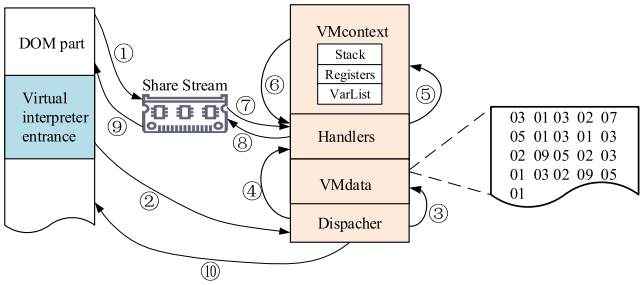


FIGURE 6. The execution process of the protected code, DOM operation are not virtualized and JSPro separate these codes. And the VM will interact with DOM using share stream. The number indicates the execution step sequences.

Algorithm 1 Virtual Interpreter Scheduling Algorithm

```
function DispatchJSVMP
InitVMcontext
bytecode ← Fetch(VMdata)
while bytecode! = END do
    HandlerNum ← Decode(bytecode)
    if HandlerNum == ret] then
        return STA.pop()
    end if
    DoHandler(HandlerNum)
end while
return result
```

3) ENCODE THE INSTRUCTIONS

As shown in Figure 4 Step ②, after completing the virtualization, we design a set of mapping rules to encode the virtual instructions. To be precise, rules are defined as a type of driver data like "03 01" in Figure 5 that can be used to get and run instructions. In the encoding process, the virtual instructions are divided into operation codes and operands to distinguish different registers or immediate data. Finally, JSpro uses these bytecodes instead of each virtual instruction, and these bytecodes will be saved in the VMdata as part of the virtual interpreter.

B. DESIGN OF VIRTUAL MACHINE PROTECTION BASED ON WEBASSEMBLY

After splitting the JavaScript statement and claiming a custom VMdata, as shown in Figure 4 step ③, we need to design the virtual machine to perform the functions represented by the VMdata. As shown in Figure 6, the virtual machine interpreter of JSPro contains four components: Dispatcher, VMdata, Handler and VMconext.

When design virtualization protection method, semantics regularly hid in the bytecode program at runtime, so it is very important for the virtual interpreter to exactly explain the semantics and restore the function of the program. Next, we briefly describe the primary workflow of the virtual machine.

As shown in Algorithm 1, the browser executes the target script file, and once the protected numerical script is

satisfied, the virtual interpreter entry function will take over the process and goes into the interpreter's workflow. After that, the virtual execution environment is initialized. VMcontext will be utilized to complete the continuation of the execution context. After entering the decoding cycle, the scheduler center fetches bytecode one by one from VMdata, then decodes it and searches for the corresponding Handler according to the mapping table. Once achieving decoding all the bytecode, it returns directly, exits the virtual interpreter, or returns the value of the stack.

1) VMCONTEXT

The classic binary virtual interpreter run in the OS native layer, which makes the stack and register could participate in the computing process. With these characteristics, the interpreter only needs a part of memory to map the real local registers. Our approach implements virtual interpreter logic of JavaScript code by C language. In other words, we need to stimulate the OS running environment in the JavaScript source code level. As Figure 6 shows, VMcontext is mainly composed of three parts, namely Stack, Registers and VarList. In general, Registers store temporary variables and VarList is a memory-like pool of variable.

2) VMDATA AND HANDLER

VMdata is a custom bytecode generated based on virtual instruction that can represent the execution logic of the original program. Handler is explanation of a custom VMdata that can be used to restore the function of the program through instructions.

We implement the entire virtual machine interpreter in C language and generate WASM through EMSCRIPTEN compilation.The design of the Handler is described in section VI-A.2. According to the virtual mapping relationship, the corresponding Handler is designed to restore the semantics of bytecode stored in VMdata. An example of the Handler is shown in Figure 7.

3) DISPATCHER

As shown in Figure 6, the Dispatcher is a key part of the virtual interpreter. It plays an important role in how to dispatch the corresponding Handler for execution. First, the Dispatcher gets the bytecode in VMdata based on VPC (program counter). Besides, it tries to execute the corresponding Handler. Lastly, the process loops until all bytecode have been explained, the Dispatcher will invoke the Handler to restore program function based on VMdata. A simple example of the structure of the Dispatcher can be found in Figure 7.

4) COMPILING AND CALLING THE VIRTUAL INTERPRETER

As shown in Figure 4 part ④, after finishing the design of the virtual interpreter, we need to compile it into WebAssembly. Figure 8 shows the process of compiling and loading the virtual machine. Compile the C file containing the virtual machine protection program into the WASM file with

```
for (;;) {
    //fetch bytecode
    byte = fetch();
    switch (byte) {
        case 1:
            //dispatch Handler
            lod_i();
            break;
        case 2:
            lod_a();
            break;
            ......
        default:
            break;
    }
}
```

(a) Dispatcher

```
void lod_i() {
    //get parameters
    int byte = fetch();
    //get numerical constants
    EMASMARGS({
        buf = $0;
    }, byte);
}
```

(b) Handler: "lod_i"

FIGURE 7. JSPro **gives code examples of** Dispatcher **and** Handler**, which illustrates the fundamental structure.**

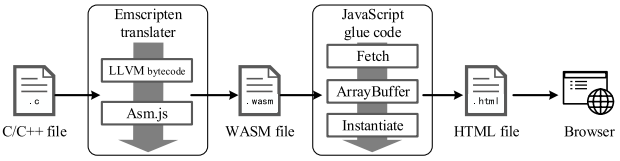


FIGURE 8. The process of compiling and loading the WebAssembly module.

EMSCRIPTEN. At the same time, the glue code for calling WASM is also generated. Finally, add the load state of the glue code, and add the call statement and the parameter transfer statement of the virtualization protection program in the original file. So the original program can successfully invoke the virtualization protection of JavaScript code.

C. SHARED STREAM FOR DOM INTERACTION

As shown in Figure 4 step ⑤, for scripts containing DOM operations, we will keep the DOM operation part in the original JavaScript, and the calculation part will be extracted and protected by virtualization. So we need to consider how the DOM element stored in JavaScript interacts with the calculation part stored in WASM after virtualization to ensure the correctness of the program. In this article, we solve this problem by using shared stream. WebAssembly can apply for a continuous memory address, we regard it as a shared stream. Specifically, when the computation in the virtual interpreter needs to be processed by the DOM element, we call the WebAssembly API functions so that deliver variables stored in DOM element to shared stream. Similarly, when the calculated value in the interpreter needs to be used by the DOM element, the value is passed to the shared stream, and the DOM element can obtain the value in the shared stream by calling the corresponding API.

VII. AN EXAMPLE

In this section, we utilize a short JavaScript code snippet as an example to illustrate how JSPro virtualizes the original code and explain how it works, as shown in Figure 6 and 9.

```
1  //The DOM element gets the user input
       variable value
2  var parm1 = document.getElementById('
       parm1').value;
3  var parm2 = document.getElementById('
       parm2').value;
4  function exchange(a, b) {
5      //Handling variables passed by DOM
       elements
6      a = a + b;
7      b = a - b;
8      a = a - b;
9  }
10 exchange(parm1, parm2);
11 //The DOM element gets the value of the
       processed variable
12 document.getElementById('parm1').
       innerHTML = parm1;
13 document.getElementById('parm2').
       innerHTML = parm2;
```

FIGURE 9. JSPro **takes JavaScript code snippet as an example to illustrate code virtualization and execution process in memory after protection.**

A. JAVASCRIPT CODE VIRTUALIZATION

As mentioned in section VI, JSPro mainly protects the computation type code, we first extract the calculation part of the code snippet and then virtualize it. Figure 5 shows the process of virtualization. As we known, JavaScript code will not be compiled ahead of time, which leads to non-semantics nature of the source code. Therefore, unlike traditional virtual protection methods, our method needs to split the instructions first. The specific virtualization process is as follows. First, we separate the computational JavaScript code into atomic operations based on the abstract syntax tree of the JavaScript code. We then translate the atomic operations to IR for getting the control flow. Finally, according to the control flow, we use IR to simulate each atomic operations for achieving equivalent translation. It's to note that to ensure the correctness of the code execution after translation, we use JavaScript to control the execution of the whole program. When running the virtualized computational JavaScript code, the control flow will be transformed to the VM which is implemented by WebAssembly.

B. RUNTIME EXECUTION

Execution of the protected code is illustrated in Figure 6. As mentioned above, our protect method to neglect the DOM part of JavaScript, and using virtual interpreter entry function instead of the calculation part after virtualization. During implementation, when executing the virtualization part, the code segment enters the virtual interpreter by calling the virtual interpreter entry function, then transmits the digital variable to shared stream following step ①. After that, following step ② and ③, the Dispatcher starts loading binary codes one by one from VMdata. When the binary codes representing the Handler number are loaded, the appropriate Handler is selected for semantic restoration, as shown in step ④. In the process of semantic reduction, the storage of intermediate results and the variable transfer between different Handlers are accomplished through VMcontext as shown in step ⑤ and ⑥. When the digital variable values received by DOM elements are needed in the process of scheduling Handler's computation, incoming values only can be obtained by accessing the shared stream through step ⑦. After the calculation is done, the result will be passed to the shared stream for DOM elements to use, as shown in step ⑧ and ⑨. The Dispatcher will load all of the bytecodes in VMdata and restored semantics by loop step ③. When all the bytecodes in VMdata are parsed, manually garbage collection of variables in WebAssembly, and then following step ⑩ to exit the virtual interpreter.

VIII. EVALUATION

A. EVALUATION SCENARIOS AND SETUP

JSPro can be evaluated in two aspects: performance and security. We will analyze its performance on three different mobile devices and four different browsers. Table 1 shows the four browsers and the corresponding operating system and hardware specifications. Table 2 summarizes the specifications of the three mobile devices. In this section, we select some valid test cases from the research [25]. Moreover, these test cases must follow these principles: ① Cover a wide variety of applications. ② Test cases are mainly related to numerical calculation. In order to obtain a relatively objective conclusion, we compared several methods that combined mainstream encryptions and obfuscation tools. Specifically, we chose four common JavaScript protection tools for comparison,including: (1) Jsonline,b10, (2) Decent Mess Up,b8, (3) Js-Obfuscator,b9 and (4) Obfuscator,b7. The results of the performance tests are illustrated below in terms of code expansion and time extension.

B. PERFORMANCE EVALUATION

In this section, we will show the time and space overhead of the test cases before and after protection.

1) TIME OVERHEAD COMPARISON

To evaluate the performance of five kinds of protection methods, we experimented with benchmark instances (selected test cases) on different browsers and different mobile devices. Before evaluation, each benchmark instance is firstly protected by JSPro and other four traditional methods respectively. Then the protected instances and the original instance run five times at each specific browser or mobile device. We calculate the average runtime of five tests as the performance. (1) Experimental results on different browsers are shown in Figure 10. It shows that the existing protection methods will lead to significant decrease in system performance. On the contrary, experimental results on benchmark instances show that JSPro performs well, in some cases, the performance even better than

TABLE 1. Browser, operating system and hardware specs.

Browser	CPU	RAM	OS	Browser Version	Browser Kernel
Firefox	Intel Core i5-3230M@2.6GHZ	4GB	Windows7	V67	Gecko
Chrome	Intel Core i5-3230M@2.6GHZ	4GB	Windows7	V64.0.3282.186	WebKit 537.36
Microsoft Edge	Intel Core i5-3230M@2.6GHZ	4GB	Windows10	V42.0.0.2741	Edge 17.17134
Safari	Intel Core i5@2.2GHZ	8GB	MacOSX10.13.3	V11.0.3	WebKit 537.36

TABLE 2. Mobile device and browser specs.

Device	CPU	RAM	OS	Browser Version	Browser Kernel
VivoXplay6L	Qualcomm Snapdragon820@2.15GHZ	6GB	FuntouchOs 7.12.24	chrome V63.0.3239.83	WebKit 537.36
Xiaomi-mix2	Qualcomm Snapdragon835@2.4GHZ	6GB	MIUI 10.3.1.0	chrome V63.0.3239.83	WebKit 537.36
iphone8	Apple Fusion@2.36GHZ	2GB	IOS 12.3.1	chrome V63.0.3239.84	WebKit 605.15

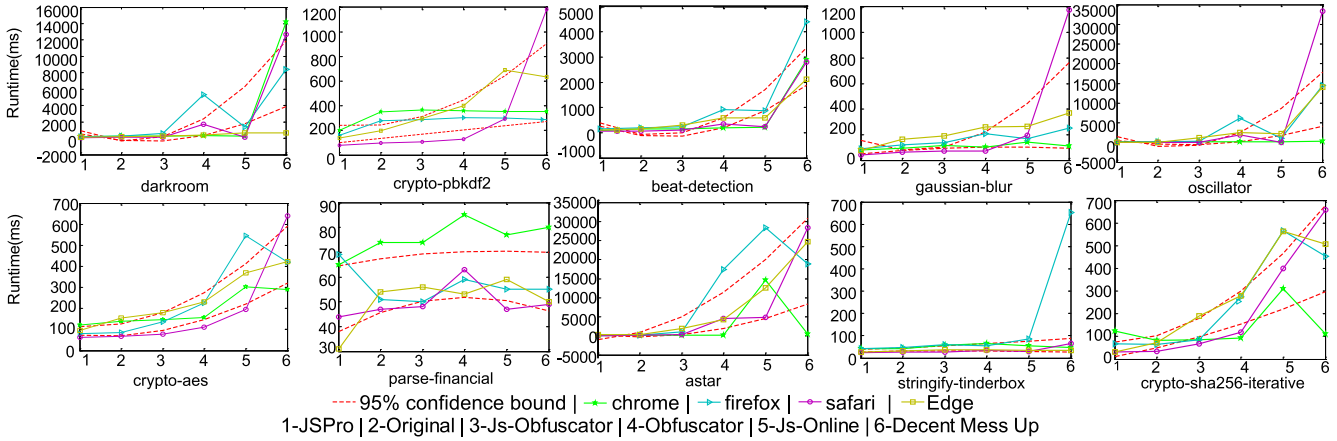


FIGURE 10. This figure shows the performance under different browsers that contain 10 cases. We can see that there are certain differences in performance of test cases under different browsers. However, traditional protection methods will cause obvious performance loss in different browsers, while our protection method `JSPro` will basically not cause performance loss, or even improve performance in some cases.

source code. (2) Further study on mobile devices with limited resources reveal that the performance of traditional protection methods is significantly declining, as shown in Figure 11. However, `JSPro`'s performance is basically consistent with source code, and even better than pre-virtualization, resulting in overall performance improvements. In particular, we test with WebAssembly-enabled browsers like `Chrome`, `Firefox`, `Safari` and `Microsoft Edge` four kinds of browsers. We also choose three kinds of devices including `Vivoxplay6l`, `Iphone8`, and `Xiaomi mix2` to evaluate the performance. In different browser performance (execution time) tests, the best performance improvement is 42.2%, the worst is 8.5%, and the average performance improvement is 15.7%. In different mobile devices, the best performance improvement is 80.1%, the worst is 9.3%, and the average performance improvement is 38.2%.

2) SIZE CHANGE COMPARISON

Figure 12 shows the comparison of size change. As we all know, classic code obfuscation always increases the program size, it is one of the negative effects. Especially for devices with limited resources, volume expansion may lead to requested operation to be not performed, or lead to insufficient storage space in system. For `JSPro`, the results are remarkable, and the size of the virtualized code is the same as the source code, or even smaller. As seen in the above experiments, under different conditions, we performed ten sets of experiments on the size increase of the protected code. The best size decreased by 61.6%, the worst size decreased by 1%, and the average size decreased by 28.2%.

C. SECURITY ASSESSMENT

Obfuscation in this work means making the code hard to understand and debug. By outlining some of the JavaScript code and compiled them down to WebAssembly, our approach makes it difficult to understand the program logic. This is because it is more difficult to track and understand the low-level compiled machine instructions than doing that for the high-level JavaScript. By translating the native machine instructions to bespoke virtual instructions, our approach forces the adversary to work on an unfamiliar instruction set. This further increases the difficulty and efforts of understanding and debugging the code. In order to evaluate the security of our system `JSPro` against a known attack, we will show

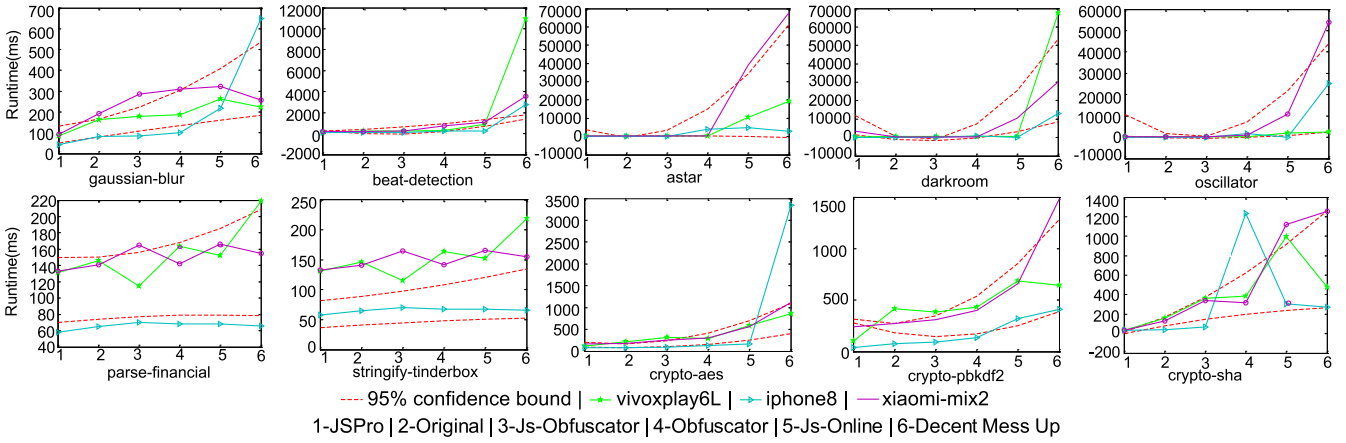


FIGURE 11. This diagram shows the performance under different mobile devices that contain 10 cases. We can see that there are some differences in the performance of test cases under different mobile devices. However, the performance loss of mobile devices with limited resources of traditional protection methods is very serious, while our protection method `JSPro` will not cause performance loss, and even can improve performance in some cases.

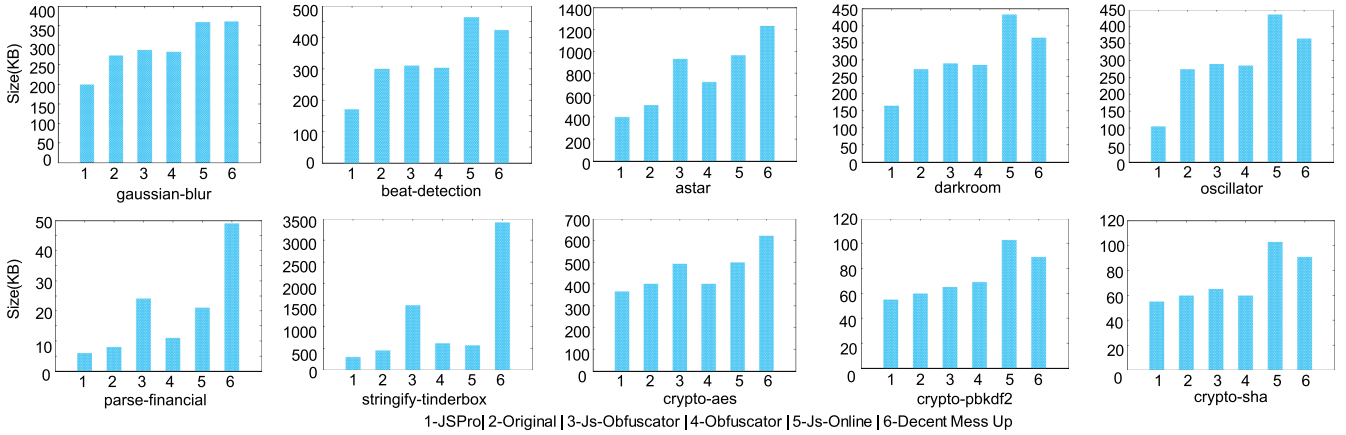


FIGURE 12. This diagram shows size change comparison that contain 10 cases.we can see that traditional protection methods will bring volume expansion, while our protection method `JSPro` can achieve smaller volume than the original program.

the reverse-engineered results on cracking tools. The specific test case is a section of the code in "Cat load", as shown in Figure 13, this code segment is regular and short, and has rich calculation process, which makes it easier for us to compare the effect of reverse analysis. We will analyze the protection results of encryption, obfuscation and `JSPro` respectively when they involve reverse cracks.

1) ENCRYPTION TOOL PROTECTION REVERSE ANALYSIS

Figure 14 (a) shows the result of the target protected code by `ToolJS` encryption. For an adversary, if he/she wants to restore the original code, some analysis tools should be first used to find the most common encryption method. In this example, the main structure of this encryption method consists of a decryption function and a section of encrypted string, and the deciphered string is decoded through the "eval ()" function. Therefore, by modifying the "eval ()" in Figure 14 (a) to "console.log ()", an adversary can get a string of compressed states, which can be formatted to the final target code, as shown in Figure 14 (b). It can be seen that the result is

```
function ears_right(cxt){
  cxt.save();
  cxt.beginPath();
  cxt.translate(canvas.width/2,canvas.height/2-100);
  cxt.rotate(-20*Math.PI/180);
  cxt.translate(-(canvas.width/2+70),-(canvas.height/2));
  cxt.moveTo(canvas.width/2-30,canvas.height/2);
  cxt.quadraticCurveTo(canvas.width/2,canvas.height/2-70,canvas.width/
  2+30,canvas.height/2); cxt.fillStyle="#228b7f";
  cxt.fill();
  cxt.closePath();
  cxt.restore();
}
```

FIGURE 13. We provide a small code snippet demonstrating the attack experiment, and three kinds of tools will be used to compare one target against another.

almost the same as source code as shown in Figure 13 exclude a partial variable name. The main logic and structure of the code have been completely restored, so it is insecurity and easy to be cracked.

As the example tell us: Hiding the "eval ()" function's body in the protected code maybe stop debugging the program, if so, the specific decryption method must be embedded to the protected code, however, it will increase

```
eval(function(p,a,c,k,e,r){e=function(c){return c.toString(a)};
if(!''.replace(/^/,String)){while(c--)r[e(c)]=k[c]||e(c);
k=[function(e){return r[e]}];e=function(){return'\\w+'};c=1};
if(k[c])p=p.replace(new RegExp('\\b'+e(c)+'\\b','g'),k[c]);return p}
('i'c(a){a.8();a.g();a.4(0.1/2,0.3/ 2-9);a.b(-7*d.e/f);a.4(-(0.1/2
+5),-(0.3/2));
a.h(0.1/2-6,0.3/2);a.j(0.1/2,0.3/2-5,0.1/2+6,0.3/2);a.k="#1";
a.m();a.n();a.o()}',25,25,'
canvas|width||height|translate|70|30|20|save|100||rotat
e|ears_right|Math|PI|180|beginPath|moveTo|function|quadraticCurveTo|
fillStyle|228b7f|fill
|closePath|restore'.split('|'),0,{}))
```

(a)Ciphertext

```
function ears_right(a){
  a.save();
  a.beginPath();
  a.translate(canvas.width / 2, canvas.height / 2 - 100);
  a.rotate( - 20 * Math.PI / 180);
  a.translate( - (canvas.width / 2 + 70), -(canvas.height / 2));
  a.moveTo(canvas.width / 2 - 30, canvas.height / 2);
  a.quadraticCurveTo(canvas.width /2,canvas.height/2-70,
  canvas.width/2+30,canvas.height/2);
  a.fillStyle = "#228b7f";
  a.fill();
  a.closePath();
  a.restore();
}
```

(b)Decryption result

FIGURE 14. Protected code using `ToolJS` encryption, we show the result of the reverse analysis in the attack experiment. As we can see, the main logic and functions are restored.

extra code size. Therefore, encryption is not an appropriate choice of JavaScript protection methods to defend an actual attack.

2) REVERSE ANALYSIS OF OBFUSCATING TOOL PROTECTION

`Js-Online` is a kind of typical code protection tool, which uses a variety of obfuscation rules. The following simply describe the obfuscation process. As Figure 15 (a) shows, First, `Js-Online` extracts `DOM` element, numeric variables and string variables. After that, split `DOM` element and string variables into multiple substrings, then put them into arrays. Second, the system uses anonymous functions with four arrays as parameters to shell the original code segment as well as replace the `DOM` elements and variables of the original code by four arrays. Besides, it also applies control flow flat method, inserts garbage code and so on to transform the original code.

Next, we will show attack process details. In this part, we give a cracked example. Just like other evaluate methods, we also choose a snippet sample to illustrate the crack procedure, and which helps to "simplify complex attack tasks". Reverse to the protection, for the obfuscation method name obfuscation, multivariate, string transfer and split, we can based on array parameters and array index which is transfer by anonymous functions to reduction `DOM` element and string variables.While the control flow obfuscation is more complex than the previous obfuscation method, the sequence of execution hidden in the array of parameters, which requires according to the index of the corresponding parameters and branches. The garbage code also can be removed by dynamic debugging and manual analysis, and the restoration of the structure takes a certain amount of time. Finally, the reverse

```
(function(i, j, k, l) {
var  m, n, o; o = n = m = i;
var  p, q, r; r = q = p = j;
var  s, t, u; u = t = s = k;
var  v, w, x; x = w = v = l;
d(n[0x9 ], k[0x4 ], j[0x1 ], v[0x5 ]), A =
  a(x[0x2 ], n[0x2 ]), B = a(j[0x6 ], t[0x1 ]),
  C = a(m[0x1 ], o[0x4 ]), D = o[0x5 ],
E = a(q[0x3], o[0x0]), F = a(k[0x3], n[0x7]),
  G = a(u[0x7],
```

(a)Name Obfuscation,Multivariate &Anonymous function self-call

```
H = a(t[ x5 ], m[0x3 ]), I = a(l[0x0 ], l
  l[0x4 ]), J = r[0x8 ]
function  a(b,a){va r
i,j,k;k  (j= b,i= j+ a);
  if(a==  = v[0x1 ]){a= v[0x1 ];return  i;}
  function  d(d,e,b,c){
  var j= w[0x1 ];var  k= x[0x3 ];for (; j;)
{switch  (k){case  13  v[0x3 ];var  l,m,q;
  q=( m= d,l= m+ e);var
r,s;s=( r= l,l= r+ a(b,c)); if(!d)
{d( o[0x6 ]); return  ;}k= n[0x8 ];
 break  ;case  n[0x8 ];return
l; j= p[0x4 ];break  ;}}}
```

(b)String transfer and split

```
m,n,o;o=(n=a(g,h,e),m=n+d(b,c,e,f));
function b(a) {
var m = l[0x1];var o = w[0x3];
for (; m;) {
switch (o) {
case i[0x8]:
a[I](-(canvas[D] / u[0x0] + k[0x8]),
  -(canvas[F]/ s[0x0]));
a[G](canvas[D] / u[0x0] - u[0x6],
  canvas[F] / s[0x0]);
a[z](canvas [D] / t[0x0], canvas[F] / s[0x0]
  - t[0x8],
canvas[D] / t[0x0] + u[0x6], canvas[F] /
  k[0x0]); a[A] = E;o = k[0x0]; break;
case v[0x3]:
case u[0x0]:
a[v[0x2]]();a[B]();a[C]();m = r[0x4];
break;}
```

(c)Control Flow Flattening, Junk code

```
})(["8b7f", "res", "Style", "ate",
[0x64, "cCur",  ![], "nPath",
  "clos","PI", Math,
 0x14],[0x2,
"ePath", "begi", "rati", "rot",
  0x1e, "mov", 0x46,  !! [], "fill", 0x0,
  "slate", "veTo", 0xb4]);
```

(d)Anonymous function self-callde

```
function b(a) {a['save']();a['beginPath']();a['translate'](canvas['idth']/2;
canvas['height']/2-100);a['rotate'](-20*Math['PI']/180);a['translate'](-
(canvas['width']/2+70),
-(canvas['height']/2));a['move'](canv['width']/2-30,canvas['height']/2);
a['quadraticCurveTo'](canvas['width']/2, canvas['height']/2-70,canvas['width']/2
+30,
canvas['height']/2); a['fillStyle']='#228b7f';a['fill']();a['closePath']
();a['restore']();}
```

(e)The result of de-obfuscated

FIGURE 15. Show the result of attack experiment aim to protected code by `JS-Online` tool. From (a) to (d) are obfuscation rules used by `JS-Online`, they will easily be removed because of its semantic maintenance. After removed these rules, we get the result of de-obfuscated which is shown in (e).

analysis result can be successfully obtained as shown in Figure 15 (b).

3) REVERSE ANALYSIS OF JSPRO PROTECTION

For the application after `JSPro` protection, there is no way to get the code segment directly after protection. As shown in Figure 16, the function content of the target program has been replaced by a function call. This function should be the entrance of the virtual interpreter. If an adversary wants to restore the logic of the target code, it first needs to enter the virtual interpreter through dynamic debugging. However, the interpreter's logic has been translated and converted into WASM module.

In the analysis process, it is found that the browser does not support debugging WASM modules and can only display independent function modules, It is difficult to get sufficient information during the dynamic debugging to enter the follow-up analysis process.

For the traditional binary code virtualization protection, there is a semantic-based anti obfuscation method, which extracts program execution sequence through a dynamic tracking the program, and further simplifies the code structure to get the final local code sequence [30]–[32] by the instruction optimization. However, this method does not apply to `JSPro`. Firstly, `JSPro` needs a combination of

TABLE 3. Factor assessment of the effect of protection method on reverse analysis.

Influence factors of reverse analysis	JS-Online	Decent Mess Up	Jsobfuscator	Obfuscator	JSPro
(1) Layout Obfuscation	Y	Y	Y	Y	Y
(2) Encoding Obfuscation	Y	N	N	Y	N
(3) Direct decryption of source code	N	Y	N	N	N
(4) Name Obfuscation	Y	N	Y	Y	Y
(5) Data Obfuscation	Y	Y	Y	Y	Y
(6) Computational obfuscation	Y	N	N	Y	Y
(7) Garbage code	Y	N	N	N	Y
(8) Control Flow Obfuscation	Y	N	N	Y	Y
(9) Logic Structure Obfuscation	N	N	N	N	Y

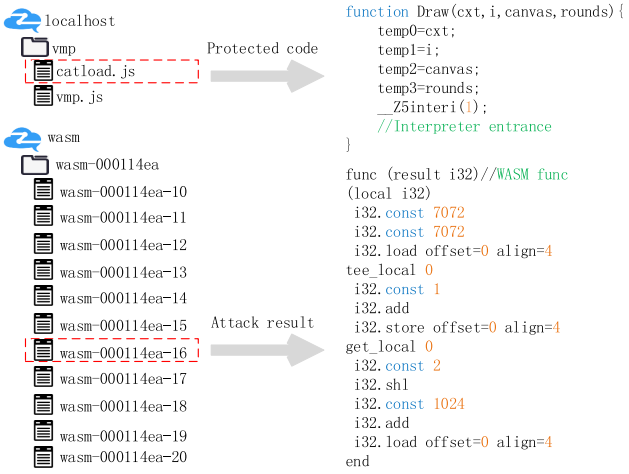


FIGURE 16. Analyzing `JSPro` protection application using `Chrome` browser. We can see that the protected code is only contain interpreter entrance,the content of interpreter cannot be debugged and the readable code disassembled by WASM is difficult to understand.

WASM and JavaScript, and the environment and language are different, making it difficult for analysts to track all the execution paths to obtain a unified instruction sequence. Secondly, in binary code virtualization protection, the target code and the interpreter is local x86 instructions, while JavaScript does not have a one-to-one instruction set corresponding to it as an explanatory language, so there is no mature simplified rule. Therefore, the reverse analysis can only draw on the traditional virtualization protection cracking process. Based on understanding the basic principle and structure of virtual machine, the method [33] is used based on the virtual execution feature, tracking and debugging the target program repeatedly, collecting the `Handler` sequence and simplifying the anti obfuscation knot which is close to the target code by manual analysis and reduction. However, which requires an attacker to have a certain understanding of the core mechanism of code virtualization protection, and the need to devote a lot of effort to manual analysis of the simplified code, will greatly increase the difficulty and cost of reverse analysis.

4) CONTRASTIVE RESULTS OF REVERSE ANALYSIS

These above analyses indicate that some related factors have an important impact on the protection strength anti-reverse engineer. We will emphasize to analyze the results shown in Table 3. As we can see, related factor(1), (4), (5), (8) are mentioned by stochastic optimization of program obfuscation [23], related factor(2), (9) are mentioned by the power of obfuscation techniques in malicious JavaScript code: A Measurement Study [34]. In addition, related factor(3), (6), (7) are customized by us where factor(3) means use encryption algorithm encrypts source code, factor(6) decomposition and the complexity of computing processes to reduce readability, and factor(7) add code that does not execute only reduces the readability of the code.

From the table, we can see that the general encryption protection has clear decryption of structured information, and the source code can be decrypted directly, and obfuscation protection will often combine a variety of protection measures to reform the structure of the code, and increase the difficulty of reverse analysis. The two kinds of methods are mainly the changes in the structure based on the original target code, and the virtualization protection of `JSPro` is to transform the target code into a custom bytecode program without the features of source code variables and control structures.

IX. DISCUSSIONS AND LIMITATION

A. SCALABILITY AND VERSATILITY

Our system `JSPro` focus on approaches that reduce protection code size and strengthen execution efficiency. Using instruction virtualization, we construct a framework based on `VM` to improve security. As a practical matter, our approach can be extended to the code including `DOM` operation, and the `DOM` operation code could be obfuscated using the original method. However, in some specific experiment designs, we also find that JavaScript code segments including many `DOM` objects are not suitable for being translated to WebAssembly. The intuition is that many `DOM` operations may involve a lot of interaction operations with WebAssembly, which results in increasing an execution latency. We will have an example to describe in details in section IX-B.

B. LIMITATION

In the following section, we discuss the limitation of the system `JSPro` concerning `DOM` operation virtualization and how they can be overcome using the browser's kernel optimization. As mentioned above, WebAssembly is designed for accelerated computation, it is not friendly to support object type. Considering the JavaScript framework, if it contains many object types, with the protection of our method, performance improvement will not be as expected. We take `DSP,b31` as an example, which is one of the JavaScript

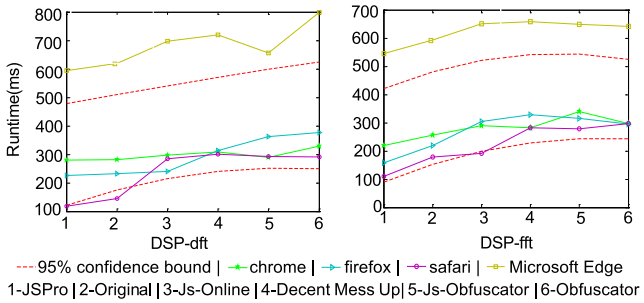


FIGURE 17. Performance of DSP with different protection methods under different browsers.

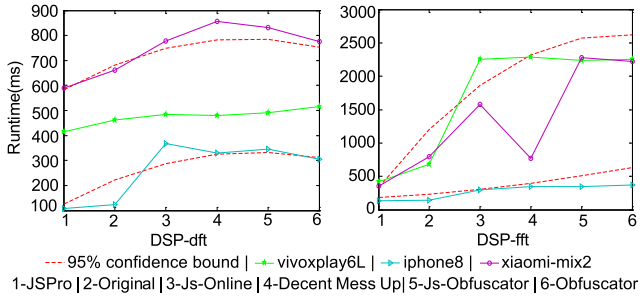


FIGURE 18. Performance of DSP with different protection methods under different mobile devices.

frameworks with the most population on GitHub. We choose two functions named ''dft'' (Discrete Fourier Transform) and ''fft'' (Fast Fourier Transformation) from the DSP to evaluate the performance of our method. The functions ''dft'' and ''fft'' contain a large number of computational code, which contributes to evaluate the performance of JSPro. The combination of Figure 17 and 18 shows the performance changes of DSP using different protection methods under different browsers and mobile devices, as Figure 17 shows traditional obfuscation can cause performance decline after protection while our method can keep performance basically unchanged or slight improved. In the best case, it can only improve 4 %, as Figure 18 shows traditional obfuscation can cause performance decline after protection but our method has better performance improvement.

However, compared to code without DOM operations mentioned above in section VIII-B.1 and VIII-B.2, the performance of DSP after JSPro protection does not meet the requirement. The next step is to optimize further the kernel of a browser to solve this problem.

X. RELATED WORK

JavaScript is widely used as an important browser language. Since the JavaScript transmission is the source code, the exposure of the source code in the front-end confrontation makes the security very fragile, so the protection of the JavaScript code has been widely concerned by the academia and the industry. Multiple commercial protection systems and platforms provide an integrated JavaScript code protection, for example, JScrambler [36] is a mature commercial protection tool that provides a variety of code protection options to compress JavaScript code, data, string segmentation coding, structure, variable name obfuscation and other protection strategies. It also has simple anti-debug measures to hinder malicious debugging analysis. JavaScript Obfuscator [37] mainly encrypts and obfuscate JavaScript code, and improves the protection of JavaScript code by encoding, encrypting and transferring strings, obfuscating variable names and inserting garbage code. However, as described earlier, the powerful debugging capabilities of the browser reduce the cost of anti obfuscation, and the obfuscation method based on semantic analysis can also reduce the cost of obfuscation. On the basis of flat control flow obfuscation and code encryption [38], JShaman [39] introduces the polymorphic mutation strategy of JavaScript code. Each call will automatically mutate and prevent the code from being dynamically debugged and analyzed. But in our analysis process, the polymorphism is achieved through the server returning different versions of the obfuscation script. So the current front-end obfuscation is based on the source code.

Recently, community has considered using the idea of code virtualization to protect programs. VMGuards [40] is a process-level VM-based code protection system. It designs two new instruction sets to protect the critical code segments and the host runtime environment where the VM runs in. [41] presents a multi-stage code obfuscation method by using improved code virtualization techniques. The key idea of this method is to iteratively obfuscate a program many times by using different interpretations, which increases the difficulty of adversaries to analyze the structure of the original program. DIVILAR [42] is a virtualization-based protection scheme to enable self-defense of Android apps against app repackaging. This is achieved by first using a diversified virtual instruction set to re-encode an Android app and then using a specialized execution engine to schedule these virtual instructions for running the protected app. Condroid [43] proposes a light-weight Android virtualization solution based on container technology. It can support multiple Android containers in a single core environment based on multiple dependent virtual machines. Unlike previous VM-based approaches, we firstly introduce a code virtualization method for JavaScript built on WebAssembly, and it can effectively enhance the protection strength with a reasonable cost of runtime overhead.

Meanwhile, there are several reverse analysis methods against virtualization protection. Reference [44] put forward a kind of statically analyzing the combination of interpreter and bytecode, reducing virtualization protection intensity. VMAttack [45] is a de-obfuscation tool for virtualization packed binaries based on automated static and dynamic analysis. The complexity of the disassembly view is notably reduced by analyzing the inner working principles of the VM layer of protected binaries. Reference [46] proposes a different approach to the problem that focuses on identifying instructions that affect the observable behavior of the obfuscated code.

XI. CONCLUSION

This paper has presented JSPro, a novel code virtualization approach for JavaScript applications. The proposed method is able to effectively protect JavaScript sensitive algorithms and data against reverse engineering attacks by adversaries, based on WebAssembly virtualized binary code. Furthermore, for computation-intensive JavaScript applications, JSPro can significantly reduce the running overhead than start-of-the arts. JSPro was evaluated on three different mobile devices and four different browsers, and the experimental results show that the protected JavaScript applications can achieve the same performance as the original program and better than the versions which are protected by other commercial obfuscation tools. We also evaluate the security of the JavaScript applications protected by JSPro through manual reverse analysis. The analysis results present that JSPro provides stronger protection than traditional JavaScript obfuscation tools but without paying the cost of significant runtime overhead. It is to note that like commercial tools, our method would decrease the performance when a JavaScript code to be protected has many DOM or interactive operations. This is also our future work on how to address this problem based on the browser engine mechanism.

ACKNOWLEDGMENT

(Shuai Wang and Guixin Ye are co-first authors.)

REFERENCES

[1] C. S. Collberg and C. Thomborson, "Watermarking, tamper-proofing, and obfuscation—Tools for software protection," *IEEE Trans. Softw. Eng.*, vol. 28, no. 8, pp. 735–746, Aug. 2002.

[2] V. van der Veen, E. Göktas, M. Contag, A. Pawoloski, X. Chen, S. Rawat, H. Bos, T. Holz, E. Athanasopoulos, and C. Giuffrida, "A tough call: Mitigating advanced code-reuse attacks at the binary level," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2016, pp. 934–953.

[3] M. G. Kang, S. McCamant, P. Poosankam, and D. Song, "Dta++: Dynamic taint analysis with targeted control-flow propagation," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, Jan. 2011, pp. 1–14.

[4] J. Bringer and H. Chabanne, "Code reverse engineering problem for identification codes," *IEEE Trans. Inf. Theory*, vol. 58, no. 4, pp. 2406–2412, Apr. 2012.

[5] M. Wu, Y. Zhang, and X. Mi, "Binary protection using dynamic fine-grained code hiding and obfuscation," in *Proc. 4th Int. Conf. Inf. Netw. Secur.*, Dec. 2016, pp. 1–8.

[6] E. Bugnion, S. Devine, M. Rosenblum, J. Sugerman, and E. Y. Wang, "Bringing virtualization to the x86 architecture with the original VMware workstation," *Acm Trans. Comput. Syst.*, vol. 30, no. 4, Nov. 2012, Art. no. 12.

[7] A. Averbuch, M. Kiperberg, and N. Zaidenberg, "Truly-protect: An efficient VM-based software protection," *IEEE Syst. J.*, vol. 7, no. 3, pp. 455–466, Sep. 2011.

[8] K. Kuang, Z. Tang, X. Gong, D. Fang, X. Chen, and W. Zheng, "Enhance virtual-machine-based code obfuscation security through dynamic bytecode scheduling," *Comput. Secur.*, vol. 74, pp. 202–220, May 2018.

[9] X. Cheng *et al.*, "DynOpVm: VM-based software obfuscation with dynamic opcode mapping," in *Proc. Int. Conf. Appl. Cryptogr. Netw. Secur.* Cham, Switzerland: Springer, 2019.

[10] Y. Dong, J. Mao, H. Guan, J. Li, and Y. Chen, "A virtualization solution for BYOD with dynamic platform context switching," *IEEE Micro*, vol. 35, no. 1, pp. 34–43, Jan. 2015.

[11] S. Jansen and A. J. Mcgregor, "Static virtualization of C source code," *Softw.-Pract. Exper.*, vol. 38, no. 4, pp. 397–416, 2010.

[12] S. Vinco, V. Guarnieri, and F. Fummi, "Code manipulation for virtual platform integration," *IEEE Trans. Comput.*, vol. 65, no. 9, pp. 2694–2708, Sep. 2016.

[13] F. Tip, P. F. Sweeney, and C. Laffra, "Extracting library-based java applications," *Commun. ACM*, vol. 46, no. 8, pp. 35–40, 2003.

[14] Z. Y. Wang and W. M. Wu, "Technique of javascript code obfuscation based on control flow tansformations," *Appl. Mech. Mater.*, vols. 519–520, pp. 391–394, Feb. 2014.

[15] M. Abdelkhalek and A. Shosha, "JSDES: An automated de-obfuscation system for malicious JavaScript," in *Proc. 12th Int. Conf. Availability, Rel. Secur.*, Aug./Sep. 2017, Art. no. 80.

[16] S. Wessel, M. Huber, F. Stumpf, and C. Eckert, "Improving mobile device security with operating system-level virtualization," *Comput. Secur.*, vol. 52, pp. 207–220, Jul. 2015.

[17] The Web. (2016). *WebAssembly*. [Online]. Available: https://webassembly.org

[18] S. Hong, J.-C. Kim, S.-M. Moon, J. W. Shin, J. Lee, H.-S. Oh, and H.-K. Choi, "Client ahead-of-time compiler for embedded Java platforms," *Softw., Pract. Exper.*, vol. 39, no. 3, pp. 259–278, 2010.

[19] P. Louridas, "Static code analysis," *IEEE Softw.*, vol. 23, no. 4, pp. 58–61, Jul. 2006.

[20] S. Mirshokraie, A. Mesbah, and K. Pattabiraman, "Guided mutation testing for JavaScript Web applications," *IEEE Trans. Softw. Eng.*, vol. 41, no. 5, pp. 429–444, May 2015.

[21] A. Barua, M. Zulkernine, and K. Weldemariam, "Protecting Web browser extensions from JavaScript injection attacks," in *Proc. 18th Int. Conf. Eng. Complex Comput. Syst.*, Jul. 2013, pp. 188–197.

[22] C. Curtsinger, B. Livshits, B. Zorn, and C. Seifert, "ZOZZLE: Fast and precise in-browser JavaScript malware detection," in *Proc. 20th USENIX Conf. Secur.*, 2011, p. 3.

[23] H. Liu, C. Sun, Z. Su, Y. Jiang, M. Gu, and J. Sun, "Stochastic optimization of program obfuscation," in *Proc. IEEE/ACM 39th Int. Conf. Softw. Eng. (ICSE)*, May 2017, pp. 221–231.

[24] K. Kuang, Z. Tang, X. Gong, D. Fang, X. Chen, T. Xing, G. Ye, J. Zhang, and Z. Wang, "Exploiting dynamic scheduling for VM-based code obfuscation," in *Proc. IEEE Trustcom/BigDataSE/ISPA*, Aug. 2016, pp. 489–496.

[25] Kraken. (2017). *Kraken Benchmark Suite*. [Online]. Available: http://krakenbenchmark.mozilla.org/

[26] NWU-IRDETO IoT & Info-Sec Joint Lab. (2017). Js-online. [Online]. Available: http://118.89.236.89:60002/login.jsp

[27] BLACKMIAOOL. (2017). *Decent Mess Up*. [Online]. Available: http://blackmiaool.com/decent-messup/playground/

[28] Tiago Serafim source code. (2017). *Javascript Obfuscator Tool*. [Online]. Available: https://javascriptobfuscator.com/

[29] Typecho. (2017). *Obfuscator*. [Online]. Available: https://obfuscator.io/

[30] B. Yadegari, B. Johannesmeyer, B. Whitely, and S. Debray, "A generic approach to automatic deobfuscation of executable code," in *Proc. IEEE Symp. Secur. Privacy*, May 2015, pp. 674–691.

[31] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "SOK: (State of) the art of war: Offensive techniques in binary analysis," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2016, pp. 138–157.

[32] B. Yadegari and S. Debray, "Symbolic execution of obfuscated code," in *Proc. 22nd ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2015, pp. 732–744.

[33] R. Rolles, "Unpacking virtualization obfuscators," in *Proc. 3rd USENIX Conf. Offensive Technol.*, 2009, p. 1.

[34] W. Xu, F. Zhang, and S. Zhu, "The power of obfuscation techniques in malicious JavaScript code: A measurement study," in *Proc. 7th Int. Conf. Malicious Unwanted Softw.*, Oct. 2012, pp. 9–16.

[35] Corbanbrook. (2010). *Dsp.js*. [Online]. Available: https://github.com/corbanbrook/dsp.js

[36] P. Fortuna and R. Ribeiro. (2018). *Jscrambler, Javascript Application Security*. [Online]. Available: https://jscrambler.com/

[37] CuteSoft Components Inc. (2017). *Javascript Obfuscator, Protects Javascript Code From Reverse Engineering*. [Online]. Available: http://www.javascriptobfuscator.com/

[38] N. Burow, S. A. Carr, J. Nash, P. Larsen, M. Franz, S. Brunthaler, and M. Payer, "Control-flow integrity: Precision, security, and performance," *Acm Comput. Surv.*, vol. 50, no. 1, p. 16, 2017.

[39] Shaman Science and Technology. (2013). *Jshaman*. [Online]. Available: http://www.jshaman.com/index.html

[40] Z. Tang, M. Li, G. Ye, S. Cao, M. Chen, X. Gong, D. Fang, and Z. Wang, "*VMGuards*: A novel virtual machine based code protection system with VM security as the first class design concern," *Appl. Sci.*, vol. 8, no. 5, p. 771, 2018.

[41] F. Hui, Y. Wu, S. Wang, and H. Yin, "Multi-stage binary code obfuscation using improved virtual machine," in *Proc. Int. Conf. Inf. Secur.*, 2011, pp. 168–181.
[42] W. Zhou, Z. Wang, Y. Zhou, and X. Jiang, "DIVILAR: Diversifying intermediate language for anti-repackaging on Android platform," in *Proc. 4th ACM Conf. Data Appl. Secur. Privacy*, Mar. 2014, pp. 199–210.
[43] W. Chen, L. Xu, G. Li, and Y. Xiang, "A lightweight virtualization solution for Android devices," *IEEE Trans. Comput.*, vol. 64, no. 10, pp. 2741–2751, Oct. 2015.
[44] J. Kinder, "Towards static analysis of virtualization-obfuscated binaries," in *Proc. 19th Work. Conf. Reverse Eng.*, Oct. 2012, pp. 61–70.
[45] A. Kalysch, J. Götzfried, and T. Müller, "VMAttack: Deobfuscating virtualization-based packed binaries," in *Proc. 12th Int. Conf. Availability, Rel. Secur.*, Aug./Sep. 2017, Art. no. 2.
[46] K. Coogan, G. Lu, and S. Debray, "Deobfuscation of virtualization-obfuscated software: A semantics-based approach," in *Proc. 18th ACM Conf. Comput. Commun. Secur.*, Oct. 2011, pp. 275–284.

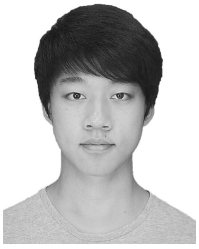
SHUAI WANG received the B.E. degree in computer science and technology from the Taiyuan University of Technology, China, in 2017. He is currently pursuing the M.S. degree in computer application technology. His current research interests include javascript security and deep learning.

GUIXIN YE was born in Tai'an, Shandong, China, in 1990. He is currently pursuing the Ph.D. degree in software engineering with Northwest University. His current research interest includes privacy and software security. He has extensive experience in authentication and software bug detection.

MENG LI received the B.E. degree in computer science and Technology from Northwest University, China, in 2017, where she is currently pursuing the M.S. degree in computer application technology. Her current research interests include wireless sensing and information security.

LU YUAN was born in 1995. She received the B.S. degree in computer science from Northwest University, China, in 2017, where she is currently pursuing the master's degree in computer science. Her current research interests include mobile computing and mobile energy efficiency.

ZHANYONG TANG received the Ph.D. degree in computer software and theory from Northwest University. He is currently an Associate Professor with the School of Information Science and Technology, Northwest University. His current research interests include network and information security, software security and protection, localization, and wireless sensor networks.



HUANTING WANG received the B.E. degree in computer science and technology from Northwest University, China, in 2018. He is currently pursuing the M.S. degree in computer application technology. His current research interests include wireless sensing and information security.

WEI WANG received the B.S. and M.S. degrees in communication engineering from Xidian University, Xi'an, China, in 2000 and 2004, respectively, and the Ph.D. degree in information and communication engineering from Northwestern Polytechnical University, Xi'an, in 2017. Her current research interests include localization algorithm, the tour planning of mobile node, and cross technology communication.

FUWEI WANG was born in Xi'an, Shaanxi, China, in 1987. He received the Ph.D. degree from Xidian University, Xi'an, in 2014. He is currently with Northwest University. His current research interests include antenna scattering, software security, RCS reduction, and metamaterials.

JIE REN was born in 1988. He received the Ph.D. degree in computer architecture from Northwest University, China, in 2017. He is currently an Assistant Professor with the Computer Science Department, Shaanxi Normal University. His current research interests include mobile computing and performance optimization.

DINGYI FANG received the Ph.D. degree in computer application technology from Northwestern Polytechnical University, Xi'an, China, in 1983. His current research interests include mobile computing and distributed computing systems, network and information security, and wireless sensor networks.

ZHENG WANG received the Ph.D. degree in computer science from The University of Edinburgh, in 2011, under the supervision of Prof. M. O'Boyle. His current research interests include parallel compilers, runtime systems, and the application of machine learning to tackle the challenging optimization problems within these areas.