Obfuscating Software Puzzle for Denial-of-Service Attack Mitigation

Yongdong Wu*, Vivy Suhendra†, Hendra Saputra‡ and Zhigang Zhao*
*Institute for Infocomm Research, Singapore
{wydong,zzhao}@i2r.a-star.edu.sg
†National University of Singapore
vivy@comp.nus.edu.sg
‡Singapore Management University
hendra_saputra@singaindo.com

***Abstract*—The software puzzle scheme counters resource-inflated Denial-of-Service (DoS) attacks by requiring each client connecting to the server to correctly solve a cryptographic puzzle before a connection can be established. It is specifically designed to thwart attempts at utilizing high-performance Graphic Processing Units (GPUs) to cut down solution time, by dynamically and randomly generating the puzzle in such a way that an attacker cannot easily translate the puzzle to a GPU implementation. The puzzle to be delivered to the client, in the form of Java bytecode, needs to be protected with code-compliant obfuscation, to hinder reverse engineering without leaking hints on wrong key attempts that the attacker can abandon quickly. The original puzzle obfuscation method permutes instructions within syntactically similar instruction sets to preserve syntactic validity regardless of the key. However, this method will not significantly obstruct a more sophisticated bytecode verification that goes beyond syntax checking. On the other hand, due to Java's stringent specifications, existing obfuscation methods that produce fully verifiable bytecode have very restricted transformations and hence weak obfuscation strength. This paper proposes an advanced Java bytecode obfuscation method with deeper consideration of bytecode validity based on JVM verification step. It overcomes the code-compliant restriction by transforming a sequence of instructions instead of individual instructions, and introduces a randomness element that enables one-to-many transformations of the software puzzle even with the same key, thus increasing the barrier to reverse engineering.**

I. Introduction

A Denial-of-Service (DoS) attack works by sending a huge number of bogus requests to the server. As the server has to spend a lot of resources in establishing connection for each of these requests, including completing SSL handshakes in the case of HTTPS servers, it may have no resources left to handle requests from legitimate clients. The client puzzle scheme [1] is a popular DoS countermeasure that increases the cost for clients to request a service, thus limiting the number of bogus requests that a DoS attacker can put through. In the scheme, the server challenges each client requesting connection with a unique cryptographic puzzle, which would take the client some time to solve. Only after the client responds with the correct puzzle solution, the server will spend the required resources to establish proper connection with the client. The puzzle is generally the inverse of a one-way cryptographic function (e.g., hash reversal), which is a hard problem requiring brute-force solution. That is, the server computes $y = \mathcal{H}(x, n)$ where $\mathcal{H}$ is a one-way function, x is a server secret, and n is a nonce for uniqueness. $\mathcal{H}$, n, and y are sent to the client, who must then solve for x by brute-force. Thus it takes much more effort for the client to solve the puzzle than it takes the server to construct the puzzle and verify the solution.

The software puzzle scheme [2] extends the client puzzle scheme to counter DoS attacks that are backed by exceptionally high computational capability in the form of many-core Graphic Processing Units (GPUs). By spreading the computational demand of client puzzles to many GPU cores, an attacker can solve enough puzzles to launch a successful DoS attack. In the software puzzle scheme, the puzzle is dynamically generated only when a client request is received by randomly combining code blocks from a code warehouse, thus preventing an attacker from preparing a faster solution implementation in advance. The code blocks consist of CPU-only instruction blocks in addition to mathematical operations as used in the client puzzle. CPU-only instructions are instructions that are not supported on GPUs or run slower on a GPU than on a CPU, such as reading local cookies, exception handling, networking function, etc. It is further proposed that the server maintains two versions of each code block: cross-platform Java bytecode to send to clients, and the more efficient C binary code for its own computation. Nevertheless, it is still possible for an attacker to create a CPU-GPU instruction mapping in order to translate the puzzle to a GPU implementation in real time. To obstruct this attempt, the puzzle code sent to clients must be obfuscated.

In obfuscating the Java bytecode, Wu et al. [2] noted that straightforward cryptographic encryption of the puzzle might undermine the scheme. When an attacker tries different key values to decrypt the puzzle in brute-force manner, decryption with the wrong key would potentially produce syntactically invalid Java bytecode, due to the random nature of standard encryption algorithms. The attacker can thus accelerate the brute-force attempt by detecting syntax violation early in the attempt and quickly abandoning the wrong key value. As such, Wu et al. adopted an obfuscation method based on substitution cipher, where each instruction is permuted over sets of syntactically similar instructions, such as the set of single-byte instructions. Given a sequence of instruction opcodes $\{o_0, o_1, ...\}$ and a key $K = \{k_0, k_1, ...\}$, the opcode o_j is replaced with $o_j + k_j \mod t$ where t is the number of instructions in the same set as o_j. Using this method, deobfuscation with any key will always result in syntactically valid code, thus eliminating the potential shortcut.

The above-described obfuscation method uses a simple definition of code validity based on operand size and value range. That is, each permutation set contains instructions with the same number of operands and whose operands have the same value range. Two such sets are described: branch instructions with 2-byte address operands, and single-byte instructions. In reality, as Java is a strongly typed language, there are more advanced details that can invalidate the bytecode. The pre-execution verification step of the Java Virtual Machine (JVM) in fact includes the following checks:

1) **Operand stack states:** If an instruction attempts to pop operands from the stack, the top of the stack at that point must contain the correct number of operands of the correct types.
2) **Local variable types:** For each local variable, value reads and stores must be of a consistent type within its live scope in the code.
3) **Local variable initializations:** A variable in the local variable array must be initialized (by storing a value into it) before it can be read.

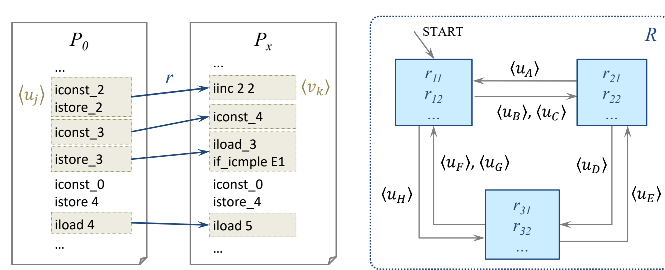
Clearly, the obfuscation outcome of instruction permutation within the set of single-byte instructions (e.g., changing `istore_0` to `fload_0`) cannot be guaranteed to pass the above bytecode validation criteria. A DoS attacker could spend a little more effort to implement on-the-fly bytecode validation that tracks the operand stack and local variable states, in order to quickly identify and discard invalid deobfuscation outcome.

At this point, it is instructive to examine existing Java bytecode obfuscation approaches for their suitability in protecting software puzzles. There are three main approaches in this respect. The first, *bytecode encryption*, is a straightforward approach where the bytecode is encrypted using a key and decrypted on the fly in JVM using a custom ClassLoader for execution. As discussed, this approach is not suitable for protecting software puzzles as it is almost guaranteed to produce invalid bytecode. The second approach, *code obfuscation*, adds complexity to the bytecode to make it harder to decompile (e.g., by control flow manipulation that results in program structures not representable with Java syntax [3]) or ways that make the decompiled code harder to read (e.g., renaming, method overloading, data/control flow obfuscation [4], [5]). The resulting code is valid and executable on standard JVM to perform its original functionality. Clearly, this cannot be used for software puzzles whose very purpose is to make the client spend brute-force effort to recover the functionality.

The third approach, *obfuscated interpretation*, aims to hide the original functionality of the code from the adversary [6] by transforming instructions in the original program to different instructions. The original software puzzle obfuscation method by Wu et al. belong to this category. The result is a valid code that executes a different task than the original code. To achieve the intended functionality, the transformation is reversed on the fly in the JVM. Existing obfuscated interpretation methods [6], [7] are FSM-based, where transformation functions are arranged into a finite state machine (FSM) to introduce a degree of randomness in determining which function is invoked at different program points. The FSM is then encrypted with a key and delivered along with the obfuscated program to the deploying device. However, these methods have very few options of transformation functions, thus low protection strength, due to the stringent criteria for code validity discussed above. Monden et al. [6] map an instruction to another instruction that has the same operand signature, and have to insert dummy instructions at the end of the code or loop to maintain a valid stack profile. In contrast, Zhang et al. [7] map an instruction to another instruction that has the same operand signature and stack behavior, but the instructions available for mapping is limited.

This paper proposes an enhanced obfuscated interpretation method suitable for the software puzzle, that preserves bytecode validity based on JVM verification requirements as described above. To expand transformation choices, the instruction transformation is not limited to one-to-one: one Java bytecode instruction can be transformed into two or more instructions, and multiple instructions can be compressed into one. This approach allows greater freedom in selecting instructions and operand values in the obfuscated code, some of which could be randomly selected from the valid range. The randomness element makes it possible to produce multiple obfuscated versions of the same software puzzle using the same secret key, and further increases the difficulty for an attacker to reverse engineer the puzzle.

II. Bytecode Transformation



(a) Bytecode encoding (b) Encoding rule set

Fig. 1. Bytecode transformation

We denote as P_0 the original sequence of bytecode instructions corresponding to the scope of a Java function, and P_x the obfuscated sequence. An encoding rule r is a function mapping a subsequence of P_0 to a subsequence of P_x, denoted informally as $\langle u_j \rangle \rightarrow \langle v_k \rangle$ for some values of j and k (Fig. 1(a)). The corresponding decoding rule r' is simply the inverse of r. The encoding rule set R (respectively, R') consists of rules r (respectively, r') arranged into an FSM similar to [6], as illustrated in Fig. 1(b). Each rule r

or r' is attached to an FSM state, and is applicable only when its attached state is active. State transitions are triggered when the encoder/decoder encounter the designated instruction sequences in P_0. The key, which the client needs to find by brute-force in the software puzzle scheme, is used to control the start state of the FSM. E.g., given key K = a string of bytes $k_0k_1...k_n$, the start state index can be calculated as $f(k_0, k_1, ..., k_n) \mod n_R$ for some function f, with n_R denoting the number of states in R. The key also controls certain transformation rules, as shall be elaborated.

TABLE I
WILDCARD DEFINITIONS

Wildcard	**Expands to**
`iconst_x`	`iconst_m1, iconst_0, ..., iconst_5`
(similarly for `lconst_x, fconst_x, dconst_x`)	
`iload_x`	`iload_0, ..., iload_3, iload`
(similarly for `lload_x, fload_x, dload_x, aload_x`)	
`istore_x`	`istore_0, ..., istore_3, istore`
(similarly for `lstore_x, ..., astore_x`)	
`ifxx`	`ifeq, ifne, ifge, ifgt, ifle, iflt`
`if_xx`	`if_icmpeq, ..., if_icmplt`
`ixxx`	`iadd, isub, imul, idiv, irem, iand, ior, ixor, ishl, ishr, iushr`
(similarly for `lxxx, fxxx, dxxx`)	

Table I lists the wildcards used for syntactically-similar groups of bytecode instructions. The decoupled wildcard notation `x` is treated as a variable, e.g., `iconst_2` is an instance of `iconst_x` with `x` = 2, `iconst_(x+1)` refers to `iconst_3`, and another instance can be denoted `iconst_y` where `y` may or may not equal `x`.

A. Instruction Transformation

For a transformation $\langle u_j \rangle \rightarrow \langle v_k \rangle$ to be valid, it must satisfy three constraints:

- It must have a deterministic inverse in order to enable correct decoding, thus it cannot involve irreversible operators such as the modulus function.
- It must preserve the stack behavior, so that it is applicable at any point of the program. That is, $\langle u_j \rangle$ and $\langle v_k \rangle$ must have the same pre-condition and post-condition regarding the stack (e.g., pop two integers and push one integer).
- Operand values appearing in $\langle u_j \rangle$ must be validly retained within $\langle v_k \rangle$, using the appropriate type and value range of operands.

Various rules can be designed based on these constraints. We present a number of examples below, but in practice, many other transformations are possible.

1) **Syntactic group transformation:** As in [7], a single instruction u can be transformed into another instruction v in the same syntactic group (Table I) except `*load_x` and `*store_x`, whose instructions are not always interchangeable for different values of `x`. Those are discussed separately below. Example:
RULE: `iconst_x` → `iconst_((x+7)%7-1)`
APPLY: `iconst_0` → `iconst_m1`

2) **Variable index permutation:** An instruction that loads from or stores to the local variable array by index can be redirected to a different index containing a variable of the same type. If there is only one variable of that type, a new variable of the required type can be added at the next available index. Example:
Let `x′` denote the next index of the same type after `x`, and suppose the local variable array contains:
`[0] ref  [1] ref  [2] int  [3] ref`
RULE: `astore_x` → `astore_x′`
APPLY: `astore_1` → `astore 3`

3) **Value masking:** Suppose $\langle u_j \rangle$ contains constant-value bytes $x_0, x_1, ..., x_m$. Then $\langle v_k \rangle$ can include instructions `bipush` or `sipush` with operands $y_0, y_1, ..., y_m$ where $y_i = (x_i \oplus z)$ where z is derived from the key K, e.g., the 7^{th} byte of K. Example:
Suppose K = `9E 30 AA 38 01 F0 64 5D 44 ...`; The value after masking is:
`(x ⊕ K[7]) = (04 ⊕ 5D) = 59`
RULE: `iload_x` → `bipush (x ⊕ K[7])`
APPLY: `iload 4` → `bipush 59`

4) **Branch instruction hiding:** This rule transforms a single conditional or unconditional branch instruction u into a sequence $\langle v_k \rangle$ containing no branch instruction, with sufficient operand bytes in $\langle v_k \rangle$ to store the branch offset bytes of u (with or without value masking). Remaining operands in $\langle v_k \rangle$ can be assigned randomly from the valid range. Example:

```
RULE:  if_xx a b → bipush (a ⊕ K[0]);
                   istore_x;
                   bipush (b ⊕ K[5]);
                   istore_y;
                   ixxx; istore_z
APPLY:  if_icmpne 00 24 → bipush 9E;
                          istore_2;
                          bipush D4;
                          istore_3;
                          idiv; istore_2
```

The wildcards `if_xx` and `ixxx` are used above for brevity, but in practice, wildcards in inputs must always be concretized so as to be invertible, while wildcards in outputs must be concretized if they play a distinguishing role. The concrete rule for this example could have, e.g., `(if_icmpeq / if_icmpne / if_icmplt / if_icmpge / if_icmpgt / if_icmple)` in place of `if_xx` and `(iand / idiv / irem / isub / ior / iadd)` in place of `ixxx`.

5) **Branch instruction insertion:** This rule transforms a non-branching sequence into a sequence containing a dummy branch instruction. While the dummy branch opcode can be randomly selected, the branch target must be chosen carefully to maintain valid stack and local variable states along the modified execution paths (discussed in Sec. III). Example:

RULE: `istore_x` → `iload_x; if_xx a b`
APPLY: `istore_2` → `iload_2;`
`if_icmpeq 00 C2`

6) **Common pattern transformation:** Common instruction sequences (usually corresponding to one source code line, e.g., a value assignment {`iconst_x`, `istore_x`}) can be transformed as a unit, to reduce code size overhead. Example:

RULE: `iconst_x; istore_y` → `iinc y x`
APPLY: `iconst_4; istore_3` → `iinc 3 4`

B. Encoding / Decoding Rule Set Construction

Each FSM state in the encoding rule set R contains a set of transformation rules selected from the possibilities explained in Sec. II-A. Correspondingly, the decoding rule set R' has a state Q' for each state Q in R, with each state Q' containing an inverse rule r' for each rule r in Q. Each state transition is labeled with a set of instruction sequences of the rule designer's choice; when the encoder encounters instructions in P_0 that matches one of the sequences, the corresponding state transition is triggered. To invert the effect of R, state transitions in R' are triggered by instructions in P_0 as well (and not P_x), that is, based on decoded instructions. The sequence matching follows the longest-match policy.

R' must be a deterministic transformation: if applying R on a given P_0 produces a number of possible P_x variants (one-to-many function), then applying R' on any P_x variant always produces the same P_0. As such, if Q contains a rule $r_1 : \langle u_j \rangle \rightarrow \langle v_k \rangle$, then Q must not contain another rule $r_2 : \langle w_m \rangle \rightarrow \langle v_k \rangle$ where $\langle w_m \rangle \neq \langle u_j \rangle$. Instruction sequences for which no rule matches are not transformed, i.e., identity transformation applies. These are implicitly present in all states of R, and have to be considered in the above restriction as well.

Fig. 2(a) gives a simple illustration where such a transformation causes incorrect recovery of P_0. The instruction sequence {`iconst_1; iload_3; if_icmpeq 00 C2`} in P_0 does not match the rule and thus, untransformed, appears as is in P_x. But the decoder finds that the inverse rule applies to the subsequence {`iload_3; if_icmpeq 00 C2`} and decodes it into {`iconst_1; istore_3`}, which deviates from P_0.

As such, we ensure that no explicit rule has overlapping output with any identity transformation: for every rule $\langle u_j \rangle \rightarrow \langle v_k \rangle$ in a state Q of R, any sequence s_0 matching $\langle v_k \rangle$ in P_0 while state Q is active is always transformed to a different sequence s_x in P_x. The transformation of s_0 can be total or partial (i.e., only a subsequence is transformed), but it cannot consist purely of syntactic group transformations (Sec. II-A, rule 1), as this does not provide enough distinction: the resulting s_x will still match $\langle v_k \rangle$. The exception is when $\langle v_k \rangle$ contains only direct opcodes and no wildcard. To formulate: for every rule $\langle u_j \rangle \rightarrow \langle v_k \rangle$ in a state of R, some subsequence $\langle w_m \rangle \subseteq \langle v_k \rangle$ must appear as input of a non-syntactic-group transformation in the same state.

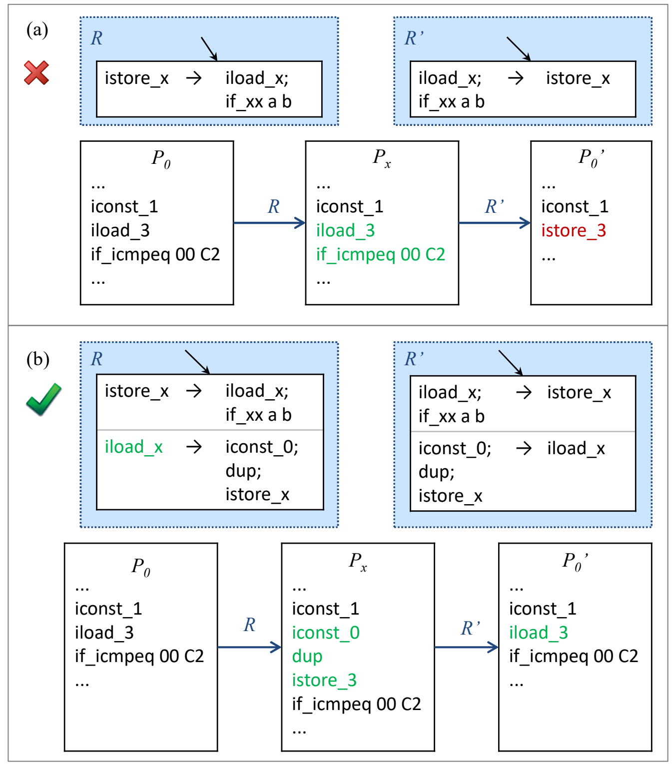


Fig. 2. Examples of improper (a) and proper (b) design of R

Fig. 2(b) shows the revised example where the rule transforming {`iload_x`} has been added to R, enabling correct recovery of P_0. Note that if the syntactic group transformation rule {`iload_x`} → {`iload_x'`} were added instead, P_0 would still not be correctly recovered. The output of the new rule also contains {`istore_x`}, which already appears in a rule input, satisfying the constraint in turn. This does introduce some input-output circularity into the design of R, though it is moderated by the fact that only a subsequence of each output needs to be transformed.

III. Encoding / Decoding Procedure

Algorithms 1 and 2 shows the encoding and decoding algorithms, respectively. For convenience, we denote the byte address of instruction u as $u.addr$; the current byte position of program P during traversal as $P.pc$; and the current stack depth of P as $P.depth$. The notation R^K refers to the encoding rule set R as controlled by the key K, for deriving the start state and value masking bytes. The start state of R^K is denoted $R^K.start$, and the active state is denoted $R^K.q$. The various aspects of the procedures are discussed in the following.

```
1  Function encode( $P_0$, $R$, $K$ )
2  // Scan through $P_0$ to collect information
3  $Vars := \varnothing$; $Targets := \varnothing$;
4  foreach load/store instruction $u$ in $P_0$ do
5      Let $(vi, vt)$: (index, type) of $u$;
6      if $\exists\ vt' \neq vt : (vi, vt') \in Vars$ then
7          // Index reuse detected
8          Create new index $vi' := P_0$.max_locals +1;
9          foreach $(vi, vt)$ load/store instruction $w$ in $P_0$ do
10             Set $w$'s index to $vi'$;
11         Add $(vi', vt)$ to $Vars$;
12     else add $(vi, vt)$ to $Vars$;
13 foreach branch instruction $u$ in $P_0$ do
14     $Targets$ += $\{(u.addr + c) \mid \forall c$: offsets in $u\}$;
15 foreach entry $e$ in $P_0$.ExceptionTable do
16     $Targets$ += { $e$.startPc, $e$.endPc, $e$.handlerPc };
17 // Transform $P_0$ into $P_x$
18 Let $w$: longest sequence length of rule inputs in $R$;
19 Initialize $B$: an empty buffer with capacity $w$;
20 $AddrMap := \varnothing$; $DepthMap := \varnothing$;
21 $R^K.q := R^K.start$;
22 foreach instruction $u$ in $P_0$ do
23     Add $u$ to $B$;
24     if $B$ is full then
25         Let $b$: the first instruction in $B$;
26         if $b.addr \in Targets$ then
27             Set $AddrMap[\ b.addr\ ] := P_x.pc$;
28         $outSeq :=$ transform( $B$, $R^K$, $Vars$);
29         Add $P_x.pc$ to $DepthMap[\ P_x.depth\ ]$;
30         Add $outSeq$ to $P_x$;
31 // End of input: transform remaining instructions in $B$
32 while $B$ is not empty do execute lines 25–30;
33 // Post-encoding adjustments
34 Perform def-use chain analysis for each variable in $Vars$;
35 Let $ND$: set of variables used without prior def;
36 foreach $(vi, vt)$ in $ND$ do
37     Insert a definition for $(vi, vt)$ at the start of $P_x$;
38 foreach branch instruction $u$ with valid offsets in $P_x$ do
39     foreach original offset $c$ of $u$ do
40         $t := AddrMap[\ (u.origAddr + c)\ ]$;
41         Modify $c$ to $(t - u.addr)$;
42 Copy $P_0$.ExceptionTable to $P_x$, adjusting addresses
     using $AddrMap$;
43 Let $DU$: def-use chains of all variables in $Vars$;
44 Let $UT : \{u.pc \mid \forall u$: unreachable instructions in $P_x\}$;
45 foreach dummy branch instruction $u$ in $P_x$ do
46     foreach unassigned offset $c$ in $u$ do
47         Let $d$: stack depth at $P_x$ after $u$;
48         $T := DepthMap[\ d\ ] - \{u.addr\} -$
             $\{t \mid pc_d \leq t \leq pc_u, (pc_d,\ pc_u) \in DU\ \}$;
49         if $T \cap UT$ is not empty then
50             Randomly choose a value $t$ from $T \cap UT$;
51             Remove $t$ from $UT$;
52         else randomly choose a value $t$ from $T$;
53         Set $c$'s value to $(t - v.addr)$;
54 Output $P_x$;
```

Algorithm 1: Encoding function

```
1  Function decode( $P_x$, $R'$, $K$ )
2  // Scan through $P_x$ to collect information
3  $Vars := \{(vi, vt) \mid (vi, vt)$ accessed in $P_x\}$;
4  $Targets := \varnothing$;
5  foreach branch instruction $u$ in $P_x$ do
6      $Targets$ += $\{(u.addr + c) \mid \forall c$: offsets in $u\}$;
7  foreach entry $e$ in $P_x$.ExceptionTable do
8      $Targets$ += { $e$.startPc, $e$.endPc, $e$.handlerPc };
9  // Skip variable initializations
10 while $u$ at $P_x.pc$ is a basic variable initialization do
11     Advance $P_x.pc$ to the next instruction;
12 // Transform $P_x$ into $P_0$
13 Initialize $P_0$: an empty bytecode sequence;
14 Let $w$: longest sequence length of rule inputs in $R'$;
15 Initialize $B$: an empty buffer with capacity $w$;
16 $AddrMap := \varnothing$; $R'^K.q := R'^K.start$;
17 foreach instruction $v$ in $P_x$ do
18     Add $v$ to $B$;
19     if $B$ is full then
20         Let $b$: the first instruction in $B$;
21         if $b.addr \in Targets$ then
22             Set $AddrMap[\ b.addr\ ] := P_0.pc$;
23         $outSeq :=$ transform( $B$, $R'^K$, $Vars$);
24         Add $outSeq$ to $P_0$;
25 // End of input: transform remaining instructions in $B$
26 while $B$ is not empty do execute lines 20–24;
27 // Post-decoding adjustments
28 foreach branch instruction $u$ in $P_0$ do
29     foreach original offset $c$ in $u$ do
30         $t := AddrMap[\ (u.origAddr + c)\ ]$;
31         Modify $c$ to $(t - u.addr)$;
32 Copy $P_x$.ExceptionTable to $P_0$, adjusting addresses
     using $addrMap$;
33 Output $P_0$;
```

Algorithm 2: Decoding function

A. Local Variable Information

To facilitate variable index permutations, the encoder (resp. decoder) performs a preliminary scan of P_0 (P_x) to identify the variable types corresponding to the local variable array indices. (The optional attribute `LocalVariableTable` of the Java class file provides this information, but is not always available.) In some cases, Java compilers may use the same index for multiple local variables with non-overlapping scopes. This is an issue if the variables are of different types. We detect such index reuse cases before transformation and re-assign the latter variable to a newly created index.

B. Byte Address Adjustment

As the transformation changes code size and hence byte addresses of instructions, we adjust all address offsets and targets that persist from P_0 to P_x (and vice versa), so that they point to the intended instructions with valid stack and variable states. The preliminary scan of P_0 or P_x collects the tar-

Function `transform(` B, FSM^K, $Vars$`)`
// Attempt rule input matching, longest first
$outSeq := \varnothing$; $inSeq := B$;
while *outSeq is empty and inSeq is not empty* **do**
  if $\exists\ \langle u_j \rangle \rightarrow \langle v_k \rangle$ *in* $FSM^K.q$: *inSeq matches* $\langle u_j \rangle$ **then**
    $outSeq :=$ instantiated $\langle v_k \rangle$ according to $Vars$;
  else remove the last instruction of $inSeq$;

// If no match, identity-transform the earliest instruction
if *inSeq is empty* **then**
  Let u: the first single instruction in B;
  $outSeq := \langle u \rangle$; $inSeq := \langle u \rangle$;

// Update buffer and effect state transition if any
$B := B - inSeq$;
while *inSeq is not empty* **do**
  Let $inSeq$: $\langle u_i : 0 \leq i < n \rangle$;
  if $\exists\ m \leq n$ *s.t.* $\langle u_i : 0 \leq i < m \rangle$ *labels a transition* $FSM^K.q \rightarrow Q$ **then**
    $FSM^K.q := Q$;
    Dequeue $\langle u_i : 0 \leq i < m \rangle$ from $inSeq$;
  else dequeue $\langle u_0 \rangle$ from $inSeq$;

Return $outSeq$;

Algorithm 3: Transform function

get addresses of branch instructions (`if*, goto*, jsr*, *switch`) after resolving the offset, as well as addresses in the `ExceptionTable` if any. During encoding or decoding, the addresses of transformed instructions corresponding to those targets are mapped accordingly, and then used for post-transformation adjustment. The address mapping is not always one-to-one as our scheme works on sequences instead of single instructions. Our policy is to map the address of the first instruction in the original sequence to the address of the first instruction in the transformed sequence.

C. Dummy Branch Offset Assignment

In selecting dummy branch targets, we consider a branching from program point a to program point b as valid if it fulfils two conditions:

- a and b must have the same stack condition (depth, operand types). For this, the encoder maintains a mapping of stack depth to program points in P_x throughout the transformation process.
- The local variable state at a must be compatible with that at b, that is, the resulting execution path containing $a \rightarrow b$ must not have variable uses without prior definition. To this end, we perform a simple intraprocedural data flow analysis after transformations to identify variable *def-use chains* [8] across all possible execution paths. Program points in between any def-use chain are ruled out as targets, as a branching to one of those points would "cut off" a chain.

Dummy branch targets are then randomly selected from among the program points that fulfill the above conditions.

D. Variable Initializations

An access to the local variable array (`*load_x`) should only be made to variables that have been initialized earlier (through `*store_x`). This could be an issue for transformations that add dummy variables for index permutation, or add originally non-existent variable access (example of rule 4). To resolve this, we reuse the variable def-use chains to identify variables for which there exists some use without prior definition. We then insert basic initializations in the beginning of P_x for all such variables. These instructions, identifiable by their position at the start of P_x, will be ignored by the decoder.

IV. Caveats

A. StackMapTable Validity

The `StackMapTable` attribute is introduced in Java 6 and becomes mandatory since Java 7 (class files version 51 and later), with the purpose of enabling faster verification [9]. This attribute specifies expected types for local variables and operand stack at selected bytecode offsets, including all branch targets in the code. As our scheme potentially changes instruction addresses and control flow, each version of the software puzzle (original, obfuscated, incorrectly deobfuscated) will likely have different `StackMapTable` entries that are incompatible with the other versions. The practical approach is to retain the `StackMapTable` compiled for the original software puzzle, which is the one legitimate client devices will ultimately execute. If the attacker checks his deobfuscation attempts against the `StackMapTable`, he will be able to tell if the attempt is wrong. Nevertheless, verifying the `StackMapTable` will still require considerable effort on the part of the attacker.

B. Runtime Error from Value Modification

Certain transformations, such as syntactic group transformations and value masking, modify constant values in the stack. The modification may cause runtime errors, e.g., if the value is later used as index to an array, where the modified value is out of the array index bound (e.g., changing 0 to -1). This error cannot be prevented during transformation, because at the bytecode level, we have no way of knowing how the values will be used later in the code. Indeed, this issue is present in all existing obfuscated interpretation methods. For the attacker to exploit this, he will need to perform deobfuscation on the fly and run (or simulate) the instructions as soon as they are deobfuscated.

V. Experimental Results

We implement our scheme in Java, using Javassist v3.20 library [10] to perform bytecode manipulation. The scheme is applied on the AES code blocks used in the software puzzle, as shown in Table II. As expected from the nature of the scheme, it incurs code size overhead more often than not. The code size overhead on the two benchmarks is up to 33.80%, while the maximum transformation runtime overhead is up to 63 ms in the worst case. The code sizes include only the

TABLE II
CODE SIZE AND RUNTIME OVERHEAD

Program	SubBytes	ShiftRows	MixCol	AddRndKey
# Transforms	15	23	75	21
P_0 size	64 B	71 B	204 B	74 B
P_x size	78 B	95 B	248 B	96 B
Size overhead	21.88%	33.80%	21.57%	29.73%
Max Runtime	31 ms	31 ms	63 ms	31 ms

(a)		(b)	
		00	iconst_0
		01	istore_3
		02	iconst_0
		03	istore 5
00	iconst_4	05	iconst_5
01	newarray 10 (int)	06	newarray 10 (int)
03	astore_1	08	astore_1
04	iconst_2	09	iinc 2 by 2
05	istore_2		
06	iconst_3	0C	iconst_4
07	istore_3	0D	iload_3
		0E	if_icmple 225 (+221)
08	iconst_0	11	iconst_ml
09	istore 4	12	istore_3
0B	iload 4	13	iload 5
0D	iconst_4	15	iconst_3
0E	if_icmpge 202 (+188)	16	bipush 68
		18	istore 4
		1A	bipush 22
		1C	istore_2
		1D	idiv
		1E	istore_2
11	aload_1	1F	aload_1
	...		...

Fig. 3. Bytecode of the sample application: (a) Original; (b) Transformed

Java functions considered in encoding and excludes other class structure components (constant pool, etc.).

We analyze a representative function `MixColumns()` in Fig. 3, which shows the original bytecode and the transformed bytecode side by side, with transformed instructions marked in boxes. The encoding rule set R and decoding rule set R' used are given in Fig. 4. The key K used is the 128-bit (16-byte) sequence: `9E 30 AA 38 01 F0 64 5D 44 27 A5 77 C0 F9 12 AC`. We see that the obfuscation method can transform the bytecode with a high level of freedom while maintaining code validity.

VI. CONCLUSION

We have described an advanced Java bytecode obfuscation method suitable for protecting software puzzles. The method maintains bytecode validity both in the obfuscated code and in the deobfuscated code regardless if the wrong key is used, hence preventing a DoS attacker from detecting the wrong key early. It solves the limitation of existing validity-preserving Java obfuscation methods by expanding transformations to instruction sequences. This approach enables a wider range of transformation options and randomness in the choice of dummy operands, making it possible to produce multiple obfuscated versions even when the same key is used, and in turn increasing the barrier for attackers to reverse engineer the software puzzle.

ACKNOWLEDGEMENT

The work was funded under the Energy Innovation Research Programme (EIRP, Award No. NRF2014EWT-EIRP002-040), administrated by the Energy Market Authority (EMA). The EIRP is a competitive grant call initiative driven by the Energy Innovation Programme Office, and funded by the National Research Foundation (NRF).

REFERENCES

[1] A. Juels and J. G. Brainard, "Client puzzles: A cryptographic countermeasure against connection depletion attacks." in *NDSS*, vol. 99, 1999, pp. 151–165.

[2] Y. Wu, Z. Zhao, F. Bao, and R. H. Deng, "Software puzzle: A countermeasure to resource-inflated denial-of-service attacks," *IEEE Transactions on Information Forensics and Security*, vol. 10, no. 1, pp. 168–177, 2015.

[3] V. Roubtsov, "Cracking java byte-code encryption," http://www.javaworld.com/article/2077342/core-java/cracking-java-byte-code-encryption.html, May 2003.

[4] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang, "On the (im)possibility of obfuscating programs," in *Advances in cryptology - CRYPTO 2001*. Springer, 2001, pp. 1–18.

[5] C. Collberg, C. Thomborson, and D. Low, "A taxonomy of obfuscating transformations," Department of Computer Science, The University of Auckland, New Zealand, Tech. Rep., 1997.

[6] A. Monden, A. Monsifrot, and C. Thomborson, "A framework for obfuscated interpretation," in *Proc. 2nd workshop on Australasian Information Security, Data Mining and Web Intelligence, and Software Internationalisation - Vol. 32*. Australian Computer Society, Inc., 2004, pp. 7–16.

[7] X. Zhang, F. He, and W. Zuo, "A framework for mobile phone Java software protection," in *Proc. 3rd Intl Conf. on Convergence and Hybrid Information Technology*, vol. 2. IEEE, 2008, pp. 527–532.

[8] U. Khedker, A. Sanyal, and B. Sathe, *Data flow analysis: theory and practice*. CRC Press, 2009.

[9] Oracle, "The Java virtual machine specification," http://docs.oracle.com/javase/specs/jvms/se7/html/index.html, February 2013.

[10] S. Chiba, "Javassist," http://jboss-javassist.github.io/javassist/.

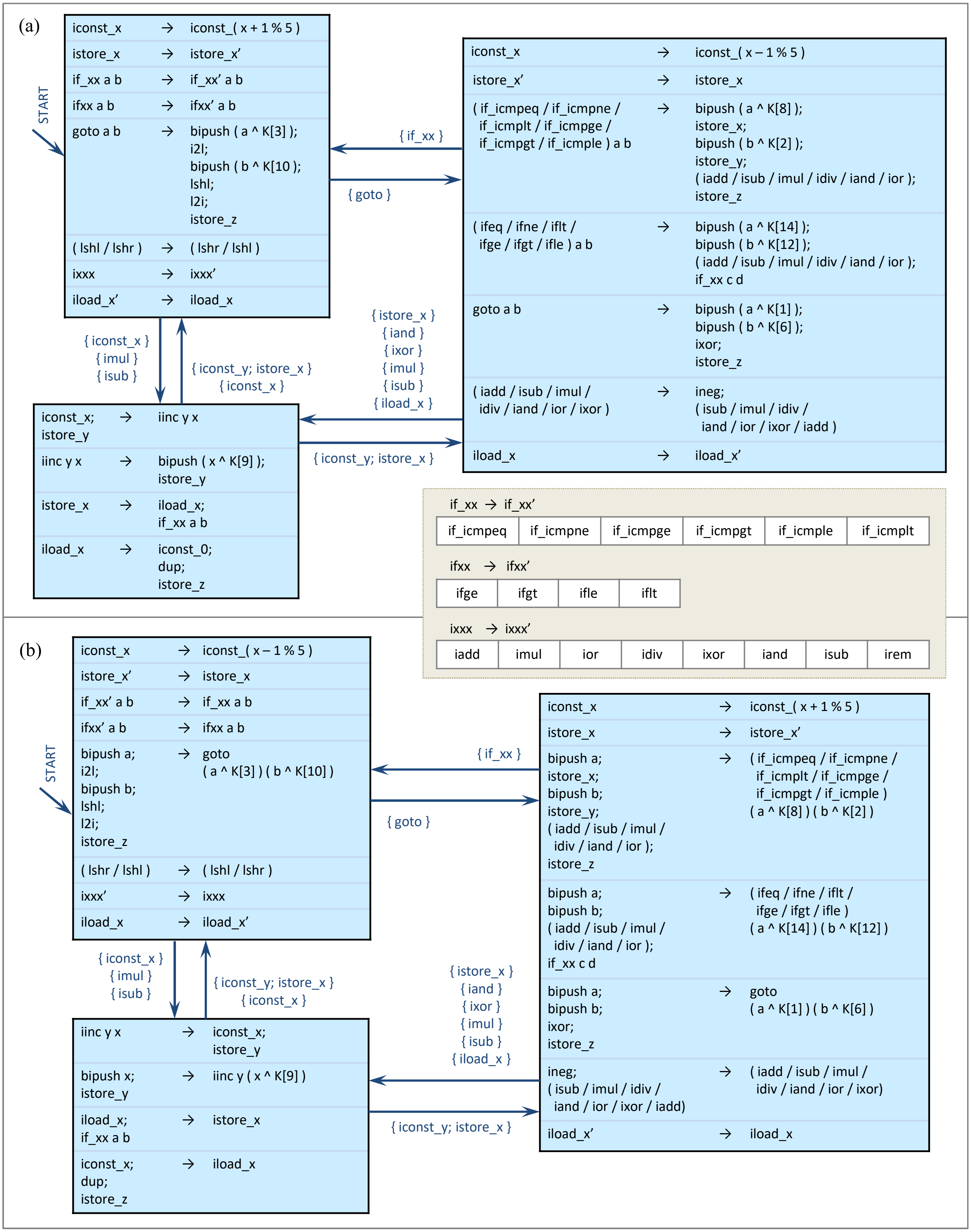


Fig. 4. The encoding rule set (a) and decoding rule set (b) used in experimental evaluation