Multi-Stage Binary Code Obfuscation Using Improved Virtual Machine

Hui Fang[1], Yongdong Wu[1], Shuhong Wang[2], and Yin Huang[2]

[1] Institute for Infocomm Research
1 Fusionpolis Way, 21-01, Singapore 138632
{hfang,wydong}@i2r.a-star.edu.sg
[2] Sumavision Soft Tech Co., Ltd.
15 Kaituo Road, Shangdi District, Beijing, 100085, China
{wangshuhong,huangyin}@sumavision.com

Abstract. A software obfuscator transforms a program into another executable one with the same functionality but unreadable code implementation. This paper presents an algorithm of multi-stage software obfuscation method using improved virtual machine techniques. The key idea is to iteratively obfuscate a program for many times in using different interpretations. An improved virtual machine (VM) core is appended to the protected program for byte-code interpretation. Adversaries will need to crack all intermediate results in order to figure out the structure of original code. Compared with existing obfuscators, our new obfuscator generates the protected code which performs more efficiently, and enjoys proven higher level security.

1 Introduction

Software obfuscation refers to transformations on the code which becomes hard to understand while preserving all functionalities. It plays an importance role in protecting confidential data and algorithms from reverse engineering or virus modification [12, 11, 22, 8]. Ideally, an adversary possessing a well-obfuscated program should be only able to learn program input/output like a black-box access. Due to this, software obfuscation has received many research interests for the last ten years [3, 33, 28, 39, 21, 24, 2, 4, 10].

The challenge in software obfuscation lies in whether or not guaranteed security and fair performance can be provided for obfuscated binary code. Specifically, code security implies resistance to static analysis and even dynamic analysis, and code efficiency implies that the obfuscated code should not run much slower than the original code. Up to now, some practical metrics for software obfuscation have been proposed in the literature [25, 21, 22, 27, 2, 9]. Meanwhile, obfuscation on Turing machine programs with formal definitions has been researched intensively as well [3, 28, 15, 42, 6, 5, 17, 7]. Unfortunately most practical obfuscation techniques lack a well-founded theoretical base, and thus it is unclear how effectively they perform. We take consideration of both practical and theoretical obfuscation metrics, and design our obfuscation algorithm align to theoretical definitions in principle.

We address the challenge by presenting an algorithm of multi-stage software obfuscation using improved virtual machine. The key idea is to obfuscate a software for many times while each time applying different interpretations in order to improve security. To fulfil the purpose, an improved virtual machine core responsible for byte-code interpretation is appended to the protected software. Under this design, an adversary must crack all intermediate results in order to figure out the structure of original code. Compared with existing obfuscators, our new obfuscator creates obfuscated code which performances more efficiently, and enjoys a higher security level.

The paper is organized as follows. Section 2 introduces the related work on software obfuscation and virtual machine. Section 3 describes our approach in two steps: block-to-byte virtual machine and multi-stage code obfuscation. Section 4 analyzes the security of our new software obfuscation algorithm. Section 5 provides experimental results. Finally, Section 6 draws a conclusion.

2 Related Work

Most existing obfuscation techniques on binary code fall into three categories:

- data transformation, such as name renaming and string encryption.
- instruction transformation, which replaces binary instructions using a library of equivalent instructions.
- control flow transformation, which transforms the graph structure of program control flow.

Data transformation does not alter program controls. Even the encrypted data will have to be decrypted inside the program for use. The code for decryption again faces the attack from reverse engineering. Therefore data obfuscation is usually applied together with other complicated obfuscation techniques to increase security [26, 16, 35].

Control flow transformation is relatively complicated [41, 18, 14, 30, 1]. Typically a control flow flattening method puts all basic blocks into a single switch statement which maintains whole control flow. It obfuscates the order in which the computations are carried out, in order to stand against static analysis. However, constant propagation on the switch variable will expose the next block to be executed. Besides, one large switch statement will generate many jumps which decreases program performance. Opaque predicates are boolean expressions whose values are known to the obfuscator but difficult for adversary to deduce. Junk codes are usually inserted into the dead path of an opaque predicate. However, for the same reason as above, there still exists risk that an adversary may figure out the value of an opaque predicate by static analysis.

Instruction transformation refers to replacement of protected binary instruction with a block of instructions which is functionally equivalent [20, 19, 23, 29, 32]. The introduced blocks representing native instruction are written as byte-codes into the program. Those byte-codes are often maintained by a virtual machine integrated with the obfuscated program. In practice, instruction transformation works well against static analysis except for runtime disassembly. However, little theoretical work has been carried out to show guarantee on its security and performance on obfuscated software.

Virtual machine (VM) based obfuscation recently becomes popular for software obfuscation, and it is probably the most sophisticated in the literature [36, 34, 32]. It usually integrates several obfuscation techniques including data permutation, instruction institution, and control flow transformation. As a result, VM obfuscation is fairly good against dynamic analysis in practice [40, 37, 31]. We observe the common way how VM obfuscator works, and summarize a general code structure for the program before and after obfuscation as shown in Figure 1. Generally speaking, a VM section will be appended to the original program, and the protected binary code will be transformed to byte-code, which is interpreted by a VM core. Finally, the entry point of the program will be redirected into VM code. To fulfil the byte-code fetching, VM core still needs to save all registers and flags in its own context, and to restore upon exiting byte-code interpretation.

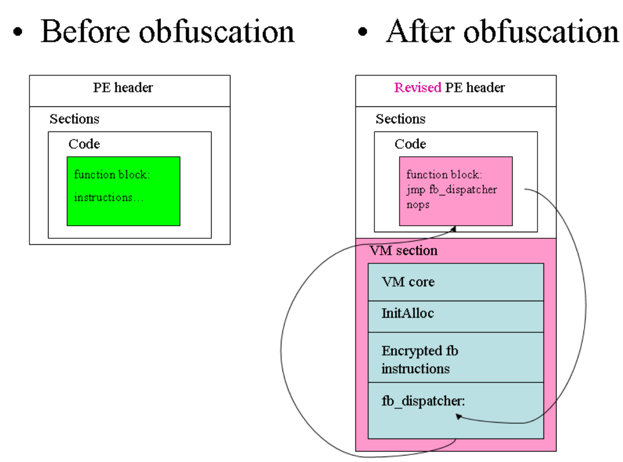

Fig. 1. Virtual machine based obfuscation.

Classical VM obfuscators suffer two drawbacks. Firstly, they generate obfuscated software which runs much slower than the original one. It is largely because of byte-code interpretation working style [37, 40]. Secondly, the security of VM obfuscated program relies merely on an uncustomized VM core integrated with program rather than each individual program. VM does not restore byte-codes to original instructions any more. Therefore success of attacking obfuscated program requires two steps: understanding VM code, and decoding mapping between binary instructions and byte-codes. One round VM obfuscation will output relatively intelligible mapping, which allows an adversary to perform instruction level analysis, and further to reconstruct the structure of original software [34, 32].

The existing works are promising under certain situations. However, the danger of software cracking is always changing and increasing [38, 24]. Therefore we propose a new approach on software obfuscation in next section, introducing a more light-weighted obfuscator which generates harder understanding codes.

3 Our Approach

In this section we firstly introduce the concept of black box security, then present new design of block-to-byte virtual machine, and describe a framework of multi-stage code obfuscation based on improved virtual machine.

A program obfuscator is often regarded as a processor on computer programs, which outputs a new program of the same functionality but with unreadable code structure [28, 10]. More precisely, a program obfuscator O is theoretically defined to be a probabilistic Turing machine or Boolean circuit, which satisfies three requirements [3]:

– (Functionality Equivalence) For every TM/circuit P and for every input $x : P(x) = O(P)(x)$.
– (Polynomial Slowdown) There exists a polynomial $q(.)$ such that for every TM/circuit P, $|O(P)| \leq q(|P|)$. TMs are additionally required that for every input x, if P halts after t steps on x then $O(P)$ halts within $q(t)$ steps on x.
– (Virtual Black Box) For any PPT A, there is a PPT oracle machine S and a negligible function $negl(.)$ such that for all TM/circuit P: $|Pr[A(O(P)) = 1] - Pr[S^P(1^{|P|}) = 1]| < negl(|P|)$.

Although Barak et al. [3] further proved that this kind of universal black box obfuscator does not exist, the theoretical concept is still useful in evaluating performance of code obfuscators. In other words, a good obfuscator shall as best as possible promise three properties: function equivalence, code efficiency, and black box security. In light of these requirements we present our customized VM obfuscator below.

3.1 Block-to-Byte Virtual Machine

The core of a virtual machine(VM) is a *dispatcher* which transforms byte-code to an implementation of binary instructions. To adapt to the purpose of program obfuscation, virtual machine must have *byte-codes* populated in and contain the *implementations* of all byte-codes for the program to protect. Specifically, a virtual machine will fetch byte-code one by one, position the target address in its *jump table*, and give control to the instruction in that address. So a complete virtual machine to be appended to the obfuscated program will be

$$V := \{Bytecodes, Impl, Jmptable, Dispatcher\}.$$

Classical VM obfuscator will map each binary instruction to a byte-code, together with its implementation (as described in Algorithm 1). We revise the design and present a block-to-byte VM obfuscation algorithm, as shown in Algorithm 2. The major difference lies in that a control flow graph (CFG) of the program is set up in prior, and then the obfuscator maps each basic block of the graph into a byte-code based on which the obfuscation is carried out.

Input: Original program P.
Output: Obfuscated program Q.
create a virtual machine V for P;
$V.Impl = \{\}$;
$V.Bytecodes = \{\}$;
for *binary instruction* $b \in P$ **do**
  translate b into byte-code B with implementation $I(b)$;
  $b =$ instruction "jump to V";
  $I(b)$'s last instruction = "jump to next to b";
  $V.Jmptable[B] = I(b)$;
  $V.Bytecodes+ = B$;
  $V.Impl+ = I(b)$;
end
output $P + V$;

Algorithm 1: Classical VM based obfuscation.

Input: Original program P.
Output: Obfuscated program Q.
construct control flow graph, $CFG(G)$;
create a virtual machine V for P;
$V.Impl = \{\}$;
$V.Bytecodes = \{\}$;
for *block* $BL \in CFG(P)$ **do**
  translate BL into byte-code B with $I(BL) = \sum_{b \in BL} I(b)$;
  BL's first instruction = "jump to V";
  $I(BL)$'s last instruction = "jump to last of BL";
  $V.Jmptable[B] = I(BL)$;
  $V.Bytecodes+ = B$;
  $V.Impl+ = I(BL)$;
end
output $P + V$;

Algorithm 2: Block-to-byte VM based obfuscation.

Figure 2 shows the format for binary instructions and VM byte-codes respectively. It also gives an example how a binary instruction was transformed into byte-code together with an implementation.

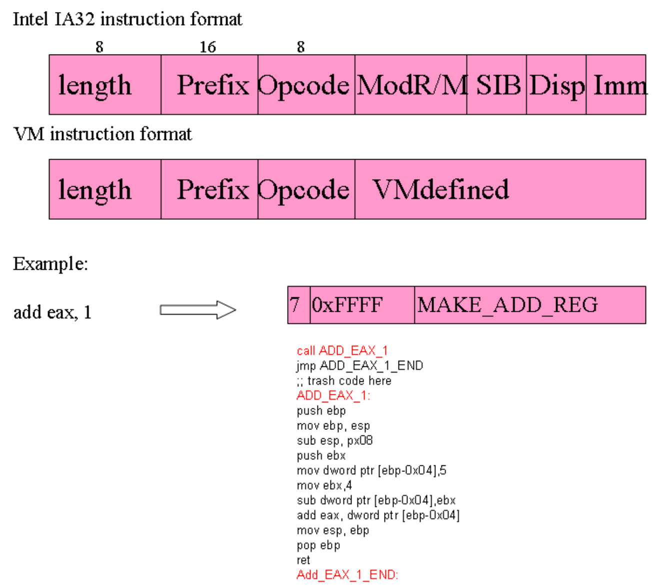

Fig. 2. Format of VM byte-code instruction and an example of implementation.

VM dispatcher works on stack based style: it saves registers for native code and create own VM stack. The return value of last execution for each byte-code was saved in VM registers (*var_RegEip* and *var_RegDI* in Figure 3) for next byte-code execution. VM dispatcher then obtains the target address by searching a jump table using byte-code as index. Target address is the location that current instruction will transfer to. VM obfuscator retrieves all target addresses of the original program in four different ways: for direct jump, target address is specified in the original instruction; for conditional jump, there are two target addresses with a predicate; for call instruction, one target address is set for called function, and another one for return address; and for return instruction, target address is stored on the stack.

3.2 Multi-Staged Code Obfuscation

In this section we extend the technique of block-to-byte virtual machine to a multi-stage obfuscation. The idea of multi-stage obfuscation algorithm is described as follows. Given an original program P, we choose a random number n to be the number of obfuscation stages, a one-way function f, and an obfuscation function Obf. Then we calculate multiple copies $\{P_0, P_1, ..., P_n\}$ of the program together with the keys $\{K_0, K_1, ..., K_n\}$ for each obfuscation stage, as shown in Figure 4.

We iteratively obfuscate program P for n times. The obfuscation key K_i is generated from each intermediate program P_i of the previous obfuscation stage, and K_i is again applied to P_i to compute P_{i+1}.

```
00401060 VM_procedure  proc near
...
004010BC VM_Entry:     ; return here upon completion of each bytecode
004010BC          inc     [ebp+var_RegEip]
004010BF          mov     eax, [ebp+var_RegEip]
004010C2          mov     al, [eax]        ; fetch one byte from pseudo-code
004010C4          mov     [ebp+var_RegDI], al
004010C7          mov     eax, offset lpJumpAddrTable
004010CC          movzx  ebx, [ebp+var_RegDI]
004010D0          shl     ebx, 2      ; x4
004010D3          add     eax, ebx           ; look up jmp table
004010D5          jmp     dword ptr [eax]  ; going to interpretation
004010D5 VM_procedure  endp
```

Fig. 3. VM byte-codes are executed by a dispatcher.

$$K_i = f(P_i),$$
$$P_{i+1} = Obf(P_i, K_i).$$

The function f maps any program into a key in binary string, satisfying that: f must have one-way hardness, and the output key can characterize the program. The examples of this type of function include: MD5 hash value of program where the program is feed as data, or the number of nodes in program's control flow graph.

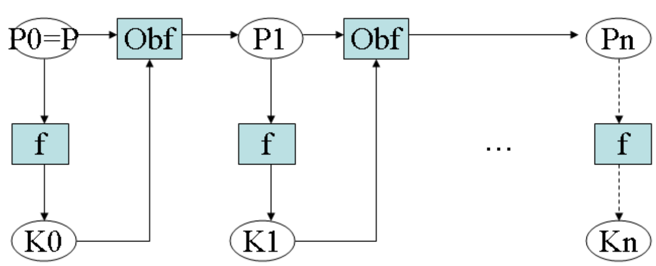

Fig. 4. The multi-stage obfuscation algorithm. P_n is output.

The obfuscation of program requires to hide program's data and/or control flow while preserving all the functionalities. In other words, each copy P_i of the program must be executable and function normally. Our idea is to extract all *jmp*/*jcc*/*call* points of P, and transform such information into a jump table. Then the jump table is obfuscated given a particular K and some dummy codes. Original program P is thus modified accordingly to jump table to preserve correct control. In other words, a separate hidden jump table will take control over program's running. Adversaries need to crack all intermediate obfuscated programs in order to recover original code's control flow.

For intra-block instructions or a single instruction, we use a revised tree structure to describe the whole process of multi-stage obfuscation. In this tree structure, each node represents a list of binary instructions (as shown in example of Figure 5). The root node x_1 refers to only one binary instruction, denoted by a circle. It links to its three children, V_1, V_2, V_3, which are different implementations of x_1. The children are called byte-codes, drawn in rectangles. Each byte-code, e.g. V_1, contains a list of binary instructions, e.g. $y_1 \rightarrow y_2 \rightarrow y_3$. In Stage-1 obfuscation, x_1 is assumed to be mapped into byte-code V_2; further in Stage-2, y_4 and y_5 of V_2 are mapped into V_5 and V_6 respectively. The path selection from an earlier stage to next stage is determined by K_i. In the example case, a formal induction of resulted code would be

$$\begin{aligned} x_1 &= V_2 \\ &= y_4 \rightarrow y_5 \\ &= V_5 \rightarrow V_6 \\ &= (z_3 \rightarrow z_4 \rightarrow z_5) \rightarrow (z_6 \rightarrow z_7) \\ &= z_3 \rightarrow z_4 \rightarrow z_5 \rightarrow z_6 \rightarrow z_7. \end{aligned}$$

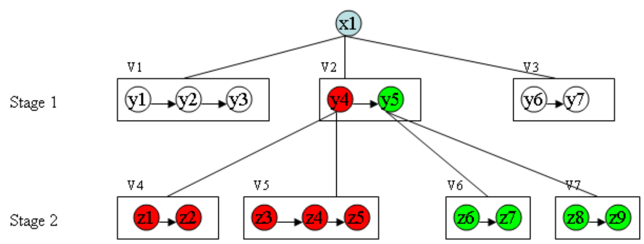

Fig. 5. Tree structure used in multi-stage obfuscation.

4 Security Analysis

This section analyzes the security of multi-stage obfuscated program in two aspects: code efficiency and black box security. Specifically we strengthen the black box security by introducing code polymorphism during multi-stage obfuscation, and improve the code efficiency by removing unnecessary jump instructions during block-to-byte VM obfuscation.

4.1 Multi-Stage Polymorphism

Polymorphism refers to that one binary instruction could have many byte-code interpretation with equivalent function. It is often used in code obfuscation to improve the difficulty in reversing program to original status.

When one instruction was obfuscated over twice, the mapping relationships from binary to byte codes become unrecognizable, due to many possible instruction combinations. Given an instruction sequence $z_3 \to z_4 \to z_5 \to z_6 \to z_7$, an adversary needs to separate them into byte-codes to understand the original program structure. In other words, one cannot easily split a sequence of instructions into correct $\{V_5, V_6\}$, and further obtain byte code V_2 which refers to x in first stage. Generally speaking, the fan-out width W of each binary node and the block size L of byte-code node for each stage determine the obfuscation complexity. In addition, the number n of stages is randomly chosen to control the complexity. The complexity of guessing increases exponentially with the number of stages. In this sense, multi-stage polymorphism makes the obfuscation of software more secure than the one obfuscated by single VM obfuscation. This claim is proved in Theorem 1.

Theorem 1. *An n-stage polymorphism tree provides $C(n)$ possible implementations for root node given constant W and L, where $C(n) = W^{L^{n-1}+\ldots+L+1}$.*

Proof. Use mathematical induction. When $n = 1$, root node links to W children which are all available choices. So $C(1) = W$ satisfies the equation. Assume $C(k) = W^{L^{k-1}+\ldots+L+1}$, and consider the case when $n = k+1$. Firstly we notice that the number of choices owned by a binary component of each stage-1 node is $C(k)$. Since each node has L components, there will be $C(k)^L$ choices for solution passing through this node. Secondly we notice that the root node can choose path from its W children. So the total possible paths will be

$$\begin{aligned} C(k+1) &= W * C(k)^L \\ &= W * (W^{L^{k-1}+\ldots+L+1})^L \\ &= W * (W^{L^k+\ldots+L^2+L}) \\ &= W^{L^k+\ldots+L^2+L+1}, \end{aligned}$$

which completes the proof. □

4.2 Improved Execution Efficiency

The classical VM obfuscator transforms protected code into byte-codes. The resulted obfuscated program then interprets byte-codes sequentially, and runs the implementation of byte-codes accordingly. However, the program control will be unconditionally switched to VM dispatcher every time when one byte-code interpretation is completed. The number of *jmp*s inserted for byte-code interpretation is proportional to the number of binary instructions. It is well known that the jump operations block the instruction streamline for execution.

In contrast, our block-to-byte VM obfuscation chooses a "basic block" to execute before jumping back to VM dispatcher. There will be no new *jmp*/*jcc*/*call* instruction inserted inside one basic block. The obfuscated program only needs to interpret bytes representing basic blocks and follows the original control flow of the program. So the number of *jmp*s inserted for byte-code interpretation is only proportional to the number of nodes in program control flow graph. By interpreting a block of instructions into only one byte-code, our multi-stage VM obfuscator is able to reduce those unnecessary jumps during code obfuscation.

The number of *jmp* instructions in the program plays a heavy part in slowing down the program execution time. Given an average block size L of control flow graph of the program, our block-to-byte VM obfuscator will generate only $\frac{1}{L}$ the number of *jmp* instructions by the classical one.

5 Experiments

The testing experiment on our multi-stage VM obfuscation module was carried out on WinXP 2.4GHz CPU and 1G RAM platform. A demo of obfuscation out is given in Appendix A. Three parameters are take into consideration: structure of control flow graph, program size, and running time of obfuscated program. We adopt IDApro [13], a disassembly tool to facilitate view on IA-32 executables. VMprotect [40], a popular VM obfuscation software, was chosen for empirical comparison.

5.1 Control Flow Graph

The complexity of a program's control flow graph reflects program intelligibility to certain extent. We capture the number of nodes and edges in graph as an indicator of graph complexity. Accordingly, the *obfuscation level* is hereafter defined as the ratio of number of nodes or edges in CFG before and after obfuscation. Table 1 presents the obfuscation level for programs using multi-stage VM obfuscation. It implies that the control flow graph becomes interleaved which leads to high obfuscation level of program.

Table 1. The number of nodes and edges of control flow graph before and after obfuscation.

Program	Original		Obfuscated		Obfuscation Level	
	#nodes,N	#edges,E	#nodes,N_2	#edges,E_2	N_2/N	E_2/E
md5	437	164	581	353	1.33	2.15
calc	458	175	746	308	1.63	1.76
draw	397	96	1439	258	3.62	2.69
crc32	151	47	354	125	2.34	2.66
aes	1908	517	3465	1392	1.82	2.70

5.2 Program Size

Program size is measured in two parameters: the number of instructions, and the size of program sections in bytes. Table 2 shows the program size of several programs before and after obfuscation. It tells that the number of instructions will normally increase at least four times after obfuscation, which implies the slowdown of obfuscated program.

Table 2. Program size before and after obfuscation.

Program	Original		Obfuscated		Increment Factor
	#instr, I	bytes	#instr, I_2	bytes	I_2/I
md5	675	1776	2837	9456	4.20
calc	485	825	2051	9559	4.23
draw	983	2109	8012	2935	8.15
crc32	231	583	1143	5665	4.95
aes	12302	32369	77748	314572	6.32

5.3 Running Time

Table 3 provides the execution time of several x86 programs on average of 10000 times. It shows that our block obfuscator generates more efficient obfuscated code than classical VM obfuscator in one stage. However when given multi-stage obfuscation, the execution time of obfuscated program increases quickly due to more complicated obfuscation.

Table 3. Execution time (secs) of obfuscated programs.

Program	Original T	VMprotect T_0	BlockVM T_1	MultiBlockVM($n = 2$) T_2	Slowdown T_2/T
md5	0.34	3.85	2.67	6.03	17.73
calc	0.12	3.40	2.34	8.73	72.75
draw	0.58	6.81	6.21	15.95	27.50
crc32	0.15	2.54	2.31	8.59	57.27
aes	0.23	4.59	5.43	11.15	48.48

6 Conclusion

We have presented a new method to obfuscate code in multiple stages to protect software from reverse engineering. The key idea is to implement a block-to-byte virtual machine to interpret byte-codes, while modifying program structure iteratively. Block obfuscation hides the binary details into byte-codes while improving the program execution efficiency; multi-stage obfuscation hides the control flow of program in a more complicated level by using a polymorphism tree. Literally, an adversary will have to decode all n variants of program to obtain the structure of original program. Meanwhile compared with classical byte-code virtual machine obfuscation, block obfuscation makes the program run more efficiently by removing unnecessary jump instructions.

Acknowledgements

This paper is sponsored by the joint research project of MOST(2010DFA11110). We are grateful to Huang Xinyi for very helpful discussions and comments.

References

1. Martn Abadi and Gordon Plotkin. On protection by layout randomization. In *23rd IEEE Computer Security Foundations Symposium*, pages 337–351, 2010.
2. Bertrand Anckaert, Matias Madou, Bjorn De Sutter, Bruno De Bus, Koen De Bosschere, and Bart Preneel. Program obfuscation: a quantitative approach. In *ACM workshop on quality of protection*, pages 15–20, 2007.
3. Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil Vadhan, and Ke Yang. On the (im)possibility of obfuscating programs. In *Crypto, LNCS2139*, pages 1–18. Springer-Verlag, 2001.
4. Philippe Beaucamps and Eric Filiol. On the possibility of practically obfuscating programs towards a unified perspective of code protection. *Journal in Computer Virology*, 3:3–21, 2007.
5. Nir Bitansky and Ran Canetti. On strong simulation and composable point obfuscation. In *Proceedings of the 30th annual conference on Advances in cryptology*, CRYPTO'10, pages 520–537. Springer-Verlag, 2010.
6. Ran Canetti and Ronny Ramzi Dakdouk. Obfuscating point functions with multibit output. In *Proceedings of the theory and applications of cryptographic techniques 27th annual international conference on Advances in cryptology*, EUROCRYPT'08, pages 489–508. Springer-Verlag, 2008.
7. Ran Canetti, Yael Tauman Kalai, Mayank Varia, and Daniel Wichs. On symmetric encryption and point obfuscation. In *7th Theory of Cryptography Conference (TCC), LNCS5978*, pages 52–71, 2010.
8. Jan Cappaert, Bart Preneel, Bertrand Anckaert, Matias Madou, and Koen De Bosschere. Towards tamper resistant code encryption. In *4th Int'l Conf. on Information Security Practice and Experience*, pages 86–100, 2008.
9. M. Ceccato, M. Di Penta, J. Nagra, P. Falcarin, F. Ricca, M. Torchiano, and P. Tonella. The effectiveness of source code obfuscation -an experimental assessment. In *The 17th IEEE International Conference on Program Comprehension (ICPC)*, pages 178–187. IEEE Computer Society, 2009.
10. Christian Collberg. Tutorial: code transformation techniques for software protection. In *ACM SIGPLAN 2009 Conference on Programming Language Design and Implementation (PLDI'09)*, 2009.
11. Christian Collberg and Clark Thomborson. Watermarking, tamper-proofing, and obfuscation - tools for software protection. *IEEE Transactions on Software Engineering*, 28:735–746, 2002.
12. Christian Collberg, Clark Thomborson, and Douglas Low. A taxonomy of obfuscating transformations. Technical report, 1997.

13. DataRescue. The ida pro disassembler and debugger, 2005. `http://www.hex-rays.com/idapro/`.
14. Jun Ge. Control flow based obfuscation. In *Proceedings of the 5th ACM workshop on Digital rights management (DRM)*, pages 83–92. ACM Press, 2005.
15. Shafi Goldweisser. On the impossibility of obfuscation with auxiliary input. pages 553–562. IEEE Computer Society, 2005.
16. Susan Hohenberger, Guy N. Rothblum, abhi shelat, and Vinod Vaikuntanathan. Securely obfuscating re-encryption. In *Theory of Cryptography Conference (TCC)*, pages 233–252. Springer, 2007.
17. Susan Hohenberger and Brent Waters. Constructing verifiable random functions with large input spaces. In *Eurocrypt, LNCS 6110*, pages 656–672, 2010.
18. Ranjit Jhala and Rupak Majumdar. Path slicing. In *Proceedings of ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 38–47. ACM, 2005.
19. Yuichiro Kanzaki, Akito Monden, and Masahide Nakamura. A software protection method based on instruction camouflage. In *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences (Japanese Edition), J87-A(6):755C767*, pages 47–59, 2004.
20. Cullen Linn and Saumya Debray. Obfuscation of executable code to improve resistance to static disassembly. In *ACM Conference on Computer and Communications Security (CCS)*, pages 290–299. ACM Press, 2003.
21. Benjamin Lynn, Manoj Prabhakaran, and Amit Sahai. Postivie results and techniques for obfuscation. In *EUROCRYPT'04*, 2004.
22. Matias Madou, Bertrand Anckaert, Bruno De Bus, and Koen De Bosschere. On the effectiveness of source code transformations for binary obfuscation. In *Proc. of the Int'l Conf. on Software Engineering Research and Practice (SERP06)*, pages 527–533, 2006.
23. Matias Madou, Bertrand Anckaert, Patrick Moseley, Saumya K. Debray, Bjorn De Sutter, and Koen De Bosschere. Software protection through dynamic code mutation. In *6th Int'l workshop on information security applications*, pages 194–206, 2005.
24. Matias Madou, Ludo Van Put, and Koen De Bosschere. Understanding obfuscated code. In *14th IEEE Int'l Conf. on Program Comprehension (ICPC)*, pages 268–274, 2006.
25. Michael Ernst Mit and Michael D. Ernst. Static and dynamic analysis: synergy and duality. In *WODA 2003: ICSE Workshop on Dynamic Analysis*, pages 24–27, 2003.
26. Akito Monden, Antoine Monsifrot, and Clark Thomborson. Security improvements for encrypted interpretation. In *Proc. 3rd Workshop on Application Specific Processors (WASP) Digest*, pages 19–26, 2004.
27. N.A. Naeem, M. Batchelder, and L. Hendren. Metrics for measuring the effectiveness of decompilers and obfuscator. In *15th IEEE Int'l Conf. on Program Comprehension*, pages 253–258, 2007.
28. T. Ogiso, Y. Sakabe, M. Soshi, and A. Miyaji. Software obfuscation on a theoretical basis and its implementation. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, E86-A(1):176–186, 2003.
29. Igor V. Popov, Saumya K. Debray, and Gregory R. Andrews. Binary obfuscation using signals. In *USENIX Security Symposium*, 2007.
30. Mila Dalla Preda, Matias Madou, Koen De Bosschere, and Roberto Giacobazzi. Opaque predicates detection by abstract interpretation. *Algebraic methodology and software technology lecture notes in computer science*, 4019:81–95, 2006.

31. Rolf Rolles. X86 virtualizer, 2008. `http://rewolf.pl/`.
32. Rolf Rolles. Unpacking virtualization obfuscators. In *Proceedings of the 3rd USENIX conference on Offensive technologies*, WOOT'09, pages 1–1. USENIX Association, 2009.
33. Benjamin Schwarz, Saumya K. Debray, and Gregory R. Andrews. Disassembly of executable code revisited. In *10th working conference on reverse engineering*, pages 45–54, 2002.
34. Monirul Sharif, Andrea Lanzi, Jonathon Giffin, and Wenke Lee. Automatic reverse engineering of malware emulators. In *Proceedings of the 30th IEEE Symposium on Security and Privacy*, pages 94–109. IEEE Computer Society, 2009.
35. Praveen Sivadasan and P. Sojan Lal. Jconsthide: a framework for java source code constant hiding. *CoRR*, 2009.
36. J.E. Smith and R. Nair. *Virtual machines: versatile platforms for systems and processes*. Morgan Kaufmann, 2005.
37. Oreans Technologies. Code virtualizer. `http://oreans.com/codevirtualizer.php`.
38. S.K. Udupa, S.K. Debray, and M. Madou. Deobfuscation: reverse engineering obfuscated code. In *12th working conference on reverse engineering*, pages 45–54, 2005.
39. Paul C. van Oorschot. Revisiting software protection. In *6th Int'l Conf. on Information Security, LNCS2851*, pages 1–13, 2003.
40. VMPsoft. Vmprotect software. `http://www.vmprotect.ru/`.
41. Chenxi Wang, Jonathan Hill, John C. Knight, and Jack W. Davidson. Protection of software-based survivability mechanism. In *Proceedings of the International Conference on Dependable Systems and Networks (formerly: FTCS)*, DSN '01, pages 193–202. IEEE Computer Society, 2001.
42. Hoetech Wee. On obfuscating point functions. In *Proceedings of the 37th annual ACM symposium on Theory of computing*, STOC '05, pages 523–532. ACM, 2005.

A Sample Output of Obfuscation

A function named *modexp* is to be obfuscated:

```
// modular exponentiation = base^exp % mod
int modexp (int base, int exp, int mod)
{
   int c = 1, expNum = 0;
   do
   {
      expNum++;
      c = (base * c) % mod;
   }
   while (expNum < exp);
   return c;
}
```

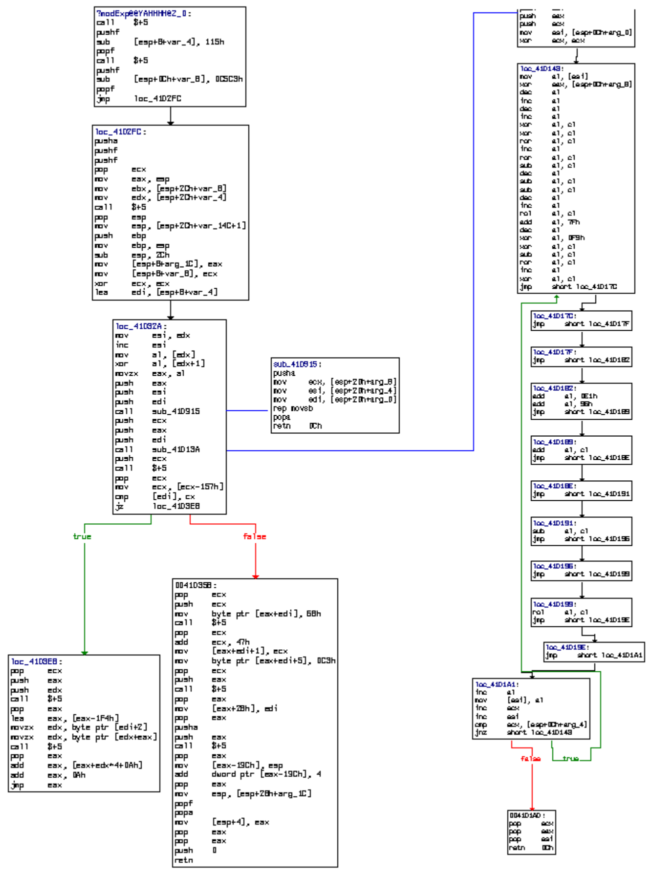

Fig. 6. CFG of obfuscated *modexp* function.