Embedded Software Security through Key-Based Control Flow Obfuscation

Rajat Subhra Chakraborty[1], Seetharam Narasimhan[2], and Swarup Bhunia[2]

[1] Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur, West Bengal, India–721302
rschakraborty@cse.iitkgp.ernet.in
[2] Department of Electrical Engineering and Computer Science
Case Western Reserve University, Cleveland, OH–44106, USA
{sxn124,skb21}@case.edu

Abstract. Protection against software piracy and malicious modification of software is proving to be a great challenge for resource-constrained embedded systems. In this paper, we develop a non-cryptographic, key-based, control flow obfuscation technique, which can be implemented by computationally efficient means, and is capable of operating with minimal hardware support. The scheme is based on matching a series of expected keys in sequence, similar to the unlocking process in a combination lock, and provides high levels of resistance to static and dynamic analyses. It is capable of protecting embedded software against both piracy as well as non-self-replicating malicious modifications. Simulation results on a set of MIPS assembly language programs show that the technique is capable of providing high levels of security at nominal computational overhead and about 10% code-size increase.

1 Introduction

The market share of embedded processors is ever-increasing, with more than 98% of the total microprocessor market share (in terms of unit sold) already occupied by them [1]. They can be found in a wide variety of electronic applications - from low-end household items such as microwave ovens to high-end 3G/4G cell phones and PDAs. Combined with this trend is the increase in computing capabilities of embedded processors (with maximum operating frequencies of up to 2 GHz in 2010) rivalling that of mainstream microprocessors [2], as they are expected to run more computation-intensive software. An example is that cutting-edge cellular devices are being increasingly used to surf the internet, play graphics intensive games and perform "mobile commerce", functionalities that were traditionally associated with personal computers. Software development for the mobile platform has also advanced immensely, with users routinely downloading, installing and using both free and commercial software for their devices.

However, this trend has increased the security concerns encompassing data confidentiality and integrity, authentication, privacy, denial of service, nonrepudiation, and digital content protection [4], which were again relevant earlier only

M. Joye et al. (Eds.): InfoSecHiComNet 2011, LNCS 7011, pp. 30–44, 2011.

in the domain of commercial and personal computing. The threat is a two-edged sword - on one hand, malicious software installed in an embedded system can harm the user; on the other hand, reverse-engineering of software causes loss of millions of dollars of intellectual property (IP) revenue to the software vendors. Unfortunately, the traditional hardware or software security measures targeting personal computers are not directly applicable to embedded systems. The computational demands of secure processing often overwhelm the computing capabilities of embedded processors, and physically the portable embedded systems are often severely constrained by form factor, resulting in limited battery capacities and memory [4].

In this work, we propose a novel technique of protecting embedded software against piracy, reverse engineering and infection by obfuscating its control-flow. The obfuscation is based on a key validation mechanism that internally generates and compares a sequence of keys with their expected values loaded from memory. The keys are execution trace dependent, meaning thereby that for different input parameters to the program, the sequence and values of keys involved in the validation process are different. The normal functionality of the program is enabled only after a successful validation process, otherwise, the program produces incorrect output. In addition, it provides additional authentication features by which even if an adversary breaks the security scheme, the ownership of the software can be proven by an *authentication* mechanism based on a *digital watermark*. The proposed technique is *not based* on the weak "security through obscurity" paradigm, where the algorithm used to obfuscate the functionality is itself hidden from the adversary [5]. In our work we assume a threat scenario where the adversary only has access to the program and tries to reverse-engineer it to unveil the security scheme, and does not have access to the hardware system which is successfully running such an obfuscated software.

The rest of the paper is organized as follows: In Section 2, we describe the proposed key-based control flow obfuscation methodology with a complete illustrative example. In Section 3, we analyze the security of the scheme against a possible attack model, and estimate the computational overhead of implementing the proposed scheme. We describe the automated flow to implement the methodology for a given MIPS assembly language program [36] in Section 4. We present the simulation results for a suite of MIPS programs in Section 5. Finally, we draw conclusions and indicate future research directions in Section 6.

2 Methodology

2.1 Obfuscation Technique

The fundamental idea of the technique proposed in this work is to validate the code during execution using a "challenge-response validation" protocol. The correct execution of the program is achieved only after the correct application of a set of input values, which constitute the *validation key sequence*. The steps of the validation process are distributed throughout the program and operates concurrently with the rest of the program, thus making it difficult to bypass

Algorithm 1. Procedure *Enumerate_Paths_Depth_First*
Enumerate all possible control-flow paths of given assembly language program segment.

Inputs: Directed Acyclic Graph G corresponding to given assembly language program segment, $instr\_stack$, $current\_node$, $last\_node$
Outputs: Set of edges ($\mathbb{E}$) with corresponding number of paths on which each edge lies

```
1: if curr_node ≠ Φ then
2:    push_on_stack(instr_stack, curr_node)
3:    if curr_node == last_node then
4:       e.pathcount ← (e.pathcount + 1) ∀ edge e on current path
5:    end if
6:    Enumerate_Paths_Depth_First(G, instr_stack, curr_node → left_child, last_instruction)
7:    Enumerate_Paths_Depth_First(G, instr_stack, curr_node → right_child, last_instruction)
8:    pop_from_stack(instr_stack)
9: else
10:   return
11: end if
```

the defense mechanism [6]. The security is also increased by the fact that the required validation key sequence depends on the input argument to the program.

The keys of the *validation key sequence* are fetched from pre-determined memory locations and compared with the expected "golden" values. If all the values match, the program execution follows the normal control flow. However, if even a single comparison fails, the program executes incorrect instructions which produces an incorrect result. The main challenge in implementing this technique is the hiding of the instructions dedicated to the validation procedure in the program. Although pre-determined values are fetched from pre-determined memory locations, the key and memory location values are not hard-coded in the program. Rather, they are derived during program execution, and different sets of values are derived depending on the input argument. This makes static analysis of the code and "program profiling" to discover the validation mechanism extremely challenging, because *each and every* validation step in the obfuscated program must be identified and neutralized to ensure that the program operates properly in *every situation*. The requirement of the predicates and variables involved in obfuscation to be *opaque*, i.e. difficult to be deduced by static analysis was pointed out in [9].

The obfuscation algorithm proceeds by finding the feasible control-flow paths in the program (or a part of it) and their dependence on the input values, and then making modifications at optimal locations in the program, such that for a given code-size and run-time overhead, the modifications would have maximum overall effect. Algorithm-1 shows the pseudo-code for the algorithm to enumerate the paths of the program using a *Depth-first Search* (DFS). The procedure assumes that the given MIPS program has been modeled as a "Directed Acyclic Graph" (DAG), with the edges forming loops removed. Each instruction of the program forms a node of the graph, and each node has one child (the non-branch instructions) or two children (the non-loop branch instructions). For each node, one among the children is always the next instruction. Note that a return from a procedure call is not treated as being part of a loop, because the "directed

Algorithm 2. Procedure *Find_Optimal_Modifications*

Find the optimal modification locations for a set of given control-flow paths and given number of modifications.

Inputs: Set of edges $\mathbb{E}$, modification pool $\mathbb{M}$, required number of modifications (M), minimum modification radius (r_{mod})
Outputs: List of modification locations in the program

```
1: Sort 𝔼 based on number of paths on which each edge e ∈ 𝔼 lies (i.e. e.pathcount)
2: num_mods ← 0
3: for all edge e ∈ 𝔼 do
4:    e.modified ← FALSE
5: end for
6: /*Iterate over the ordered edges and make modifications based on r_mod constraint*/
7: for i = 1 to |𝔼| and num_mods < M do
8:    Set 𝔼_r = {e_j ∈ 𝔼 : |e_i − e_j| ≤ r_mod} /* |e_i − e_j| stands for the physical separation of the two
      edges */
9:    if e.modified == FALSE ∀e ∈ 𝔼_r then
10:      Choose previously unchosen m ∈ 𝕄
11:      Insert m on e_i
12:      e_j.modified ← TRUE ∀e_j ∈ 𝔼_r
13:      num_mods ← num_mods + 1 /*Update number of modifications*/
14:   end if
15: end for
```

acyclic" nature of the graph can be still maintained. In addition to the regular DFS, the number of paths on which an edge lies is tracked. This information is utilized in determining optimal locations to perform modifications in the program, as described next.

Algorithm-2 shows the procedure to find the optimal locations to make M modifications for a given program (or a part of it). At first, the edges of the graph are ranked in descending order in terms of the number of paths on which the edges lie. Then, M modifications chosen greedily from a pool of modifications are inserted on the top-ranked edges, with the constraint that the modified edges are situated at least a pre-defined "modification radius" r_{mod} distance away from each other. If any edge connects two vertices which do not represent consecutive instructions in the program, jump instructions are used to connect the modification code block to the two vertices on the edge. The following points should be noted about this algorithm:

- Choosing the top-ranked edges ensures maximum effect of a single modification on multiple paths, while the r_{mod} constraint ensures that the modifications are not inserted too close to each other.
- The constraint r_{mod} determines the *average number of modifications per path*:

$$M_{av} = \frac{\sum_{i=1}^{|\mathbb{P}|} M_i}{|\mathbb{P}|} \tag{1}$$

  where $|\mathbb{P}|$ denotes the total number of paths in the part of the program segment being processed, M_i denotes the number of modifications lying on the i-th path, and $1 < M_{av} \leq M$. An increase in the value of M_{av} can

be thought of to signify an increase in the security of the system, because more successful validations are required on average per path to make the program run successfully. Another metric that is determined by r_{mod} is the *average distance between modifications*. Let $\mathbb{E}_{mod}$ be the list of modified edges, ordered by their positions in the program, and M be the total number of modifications inserted. Then the *average distance between modifications* is given (for $M > 1$) by:

$$D_{av} = \frac{\sum_{i=1}^{M-1} |e_{i+1} - e_i|}{M-1} \tag{2}$$

for $e_i \in \mathbb{E}_{mod}$, with $r_{mod} \leq D_{av} < \frac{N}{M-1}$, where N is the number of instructions in the program. If r_{mod} is small, say $r_{mod} = 1$, the minimum value possible, the top M ranked edges would be chosen which would increase the value of M_{av}. However, on the flip-side, the value of D_{av} might decrease, meaning that the modifications would be placed too close to each other which puts them at the risk of being more identifiable to an adversary. Also, a higher value of M_{av} also implies an increase in the average execution time of the obfuscated program with respect to the original program. Hence, the parameter r_{mod} provides a degree of freedom to balance between the quantitative metrics M_{av} and D_{av}, and the performance of the program.

- This algorithm inserts the modifications at "preferred pseudo-random" locations, with preference being given to locations that would affect the maximum possible number of paths, while being "pseudo-random" in the sense that the modification locations are distributed throughout the program, through the effect of r_{mod}.
- If a modification is inserted between two instructions which are part of a loop, then the key-validation step would be repeated as many times as the loop repeated, even if the validation is successful. To avoid this, the modification should be such that any successful validation is "remembered", so that the next time the loop is executed, the validation mechanism is not exercised. This can be implemented easily by having a "flag" register and local jumps in the modification. We have elucidated this point with an example in the next sub-section.
- To increase the level of security, the operations dedicated to deriving and comparing the keys of a sequence do not appear in the order in which the keys are compared.

Next we give a complete example program to elucidate the two algorithms described above.

2.2 Obfuscation Example

Fig. 1(a) shows an example MIPS assembly language program to calculate and display the value of the n-th Fibonacci number for a given non-negative integer

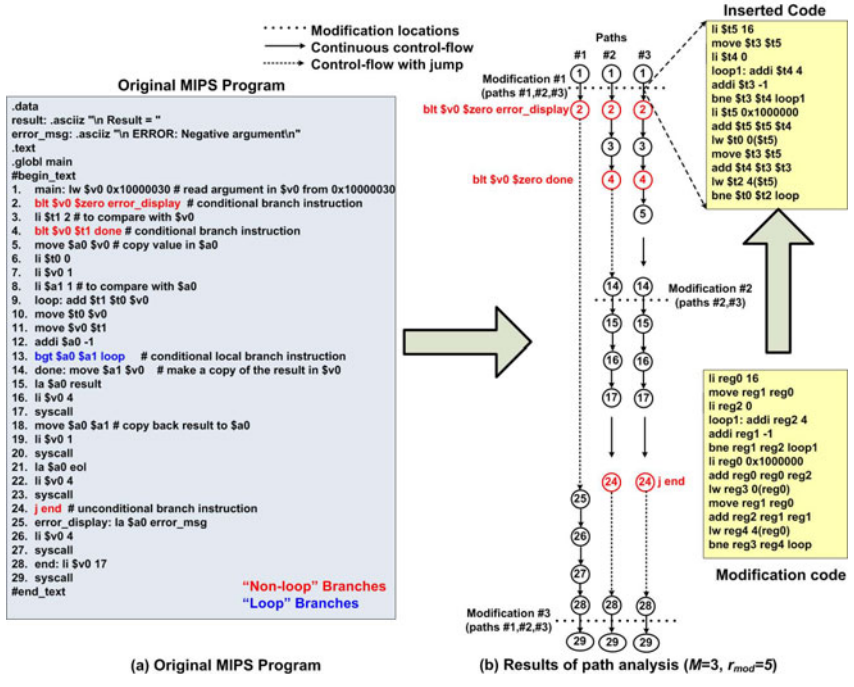

Fig. 1. Example of application of the proposed algorithm on a MIPS program to calculate the value of the n-th Fibonacci number for a given non-negative integer n

n. The main part of the program to be modified occurs between the markers ***#begin_text*** and ***#end_text***, and the instructions between these two markers have been numbered for ease of understanding. The DAG representation of the program has been shown in Fig. 1(b). The feasible control paths of the program are then enumerated by analyzing the DAG using Algorithm-1. The feasible paths for this program (paths #1, #2 and #3) are shown in Fig. 1(b), where each instruction has been represented by its serial number. Note that the different paths are followed depending on the value of the input argument n to the program - path-1 if $n < 0$, path-2 if $0 \leq n < 2$ and path-3 if $n \geq 2$. When Algorithm-2 is applied to find the optimal modification locations for $M = 3$ modifications and $r_{mod} = 5$, the modifications are placed between instructions 1 and 2 (modification #1), between 14 and 15 (modification #2) and between 28 and 29 (modification #3). Modification #1 and #3 affect all three paths, while modification #2 affects only paths 2 and 3. The average number of modifications is per path is thus $M_{av} = (3+3+2)/3 = 2.67$, which is less than the ideal value of $M_{av} = M = 3.00$. The average distance between modifications $D_{av} = 9.00$, while the ideal value is $\frac{N}{2} = \frac{29}{2} = 14.50$.

Note that Algorithm 2 implies that the first modification would always be inserted on one of the edges connecting the "root node" to the node corresponding to the first branch instruction in the program. This feature might make the first

modification identifiable to an adversary performing static analysis. This issue is handled by modifying the algorithm so that an exception is made about the position of the first modification, so that no modification appears between the "root" node and the first branch node.

An example modification has also been shown which is derived from the corresponding *modification pool* after binding the generic register names $reg0$, $reg1$ etc. to actual resisters $t5$, $t3$, etc. As mentioned before, the register binding keeps the original functionality of the program functionally correct by a *liveness analysis*. In the given case, registers $t0$ and $t2$ collect the input and golden values of the key from memory locations 0x10000040 and 0x10000044 respectively, and normal operation is allowed only if the fetched values match. In this particular case, incorrect operation is due to the fact that the register $t0$ contains an incorrect value (it should contain zero when the label *loop* is reached). In case no registers are found free to be used bound to generic registers, register spilling and restoration has to be applied.

2.3 Implementation

To make the obfuscated software operate correctly, the user must buy the software license in the form of a small support software from the software vendor to manage the key installation in memory. The user has to run this support software to install the keys in the correct memory location, and then install the main software. The security of the scheme can be increased by changing the key sequence for each instance of the licensed software, so that the support software would be bound with the particular copy of the original software which it was designed to activate.

2.4 Integration with Hardware-Assisted Approaches

The proposed software obfuscation technique can co-exist with hardware-assisted security solutions, such as *Trusted Platform Module* (TPM) [17,18], thus adding an extra level of protection. The security features provided in such platforms can be useful in situations where the adversary has physical access to the hardware successfully running the program. In addition to the proposed software obfuscation technique, if the memory contents are encrypted (e.g. in [15]) or memory addresses are re-mapped to hide the control flow (e.g. in [27]), the adversary would face an additional challenge of first breaking the hardware-assisted security scheme, and then de-obfuscating every obfuscated software individually.

3 Obfuscation Efficiency and Overheads

In this section we present theoretical analyses to obtain a quantitative estimate of the achievable security and overhead incurred by the proposed scheme.

3.1 Obfuscation Efficiency

We borrow the following metrics which have been previously proposed to estimate the success of a software obfuscation scheme [9]:

- *Potency*: the complexity in comprehending the obfuscated program compared to the unobfuscated one.
- *Resilience*: difficulty faced by an automatic de-obfuscator in breaking the obfuscation.
- *Stealth*: how well the obfuscated code blends in with the rest of the program, and
- *Cost*: how much computational overhead it adds to the obfuscated program.

A potent software obfuscation technique should provide high levels of *potency*, *resilience* and *stealth*, while incurring minimal *cost*. In particular, it should provide sufficient protection against both *dynamic* (i.e. run-time) and *static* program analyses. The technique automatically provides high levels of protection against dynamic analysis because of the fact that the particulars of the basic "challenge-response" mechanism of fetching the key from memory, comparing it with the golden key, and modifying the control-flow based on the result of the comparison, vary depending on the input arguments of the program. Because the input argument-space of most practical programs is larger beyond complete enumeration, hence, breaking the obfuscation scheme simply by observing the execution of the obfuscated program is practically infeasible. Hence, we concentrate on the protection provided by the proposed key-based obfuscation methodology against static code analysis efforts of an adversary.

Consider an assembly language program containing N instructions, to which n instructions are added to modify the control flow by the technique described above, as a result of which the code size increases to $(N + n)$. Let there be L "load" instructions in the original program, to which l "key load" instructions are added during modifications to increase the number of load instructions to $(L + l)$. Note that as pointed out earlier, these load instructions need not occur in the same order as the key comparison sequence. Similarly, let there be C "comparison-based branch" instructions in the original program to which c are added to bring the total number of branch instructions to $(C + c)$. To identify the modifications that have been made to the original program based on random choice, an adversary must perform the following steps:

- Identify the n instructions dedicated in modifying the original program, out of a total $(N + n)$ instructions in the obfuscated program. This is one out of $\binom{N+n}{n}$ possibilities.
- Identify the l "load" instructions dedicated to the obfuscation scheme out of the total $(L + l)$ "load" instructions, and from them determine the correct order in which the keys are collected from memory and compared to modify the control flow. Note that the adversary does not know a-priori the number of key comparisons for a given feasible control-flow path of a given program.

Let M_{av} be the average number of modifications performed among all the feasible control-flow paths of the given program. Then, to break the scheme, the adversary has to make exactly one out of $\left[\sum_{i=1}^{\lceil M_{av} \rceil} \binom{L+l}{i} \times i!\right]$ choices to determine the correct number and sequence of keys to be applied.

- Identify the c "comparison-based branch instructions" dedicated in control-flow modification, from a total of $(C+c)$ such instructions in the obfuscated program.
- Identify the $(n-l-c)$ dataflow operations dedicated to obfuscate the code, from among the total $(N+n-L-C-l-c)$ in the obfuscated code.

Combining the three above factors, we propose the following quantitative metric to estimate the effectiveness of the proposed key-based obfuscation scheme:

$$M_{obf,random} = \frac{1}{\left[\sum_{i=1}^{\lceil M_{av} \rceil} \binom{L+l}{i} \times i!\right] \times \binom{C+c}{c} \times \binom{N+n-L-C-l-c}{n-l-c}} \tag{3}$$

Lower values of this metric implies higher levels of *potency*, *resilience* and *stealth*. To get an idea of the numerical order of this metric, consider the example shown in Fig. 1 and the portion of the code between the two markers ***#begin_text*** and ***#end_text***. Assuming the length of all modifications to be similar to the one shown, we have the values $\lceil M_{av} \rceil = 3$, $r_{mod} = 5$, $N = 29$, $n = 3 \times 13 = 39$, $C = 3$, $c = 3 \times 2 = 6$, $L = 1$ and $l = 3 \times 2$. This gives the value $M_{obf} \approx 9.63 \times 10^{-20}$. In real-life applications, the value of this metric would be much smaller because of larger values of N and L, which in turn would allow larger values of n and l.

3.2 Computational Overhead of the Obfuscation Technique

Time Complexity. The time complexity of the path enumeration step is essentially the time complexity of the dept-first traversal, which is $O(|\mathbb{V}| + |\mathbb{E}|)$, where $|\mathbb{V}|$ and $|\mathbb{E}|$ are the number of vertices and edges respectively in the graph [32]. However, note that in our particular case, $N - 1 \leq |\mathbb{E}| \leq 2N$, where $N = |\mathbb{V}|$ is the number of instructions in the block of the program to be obfuscated. The lower limit occurs when there is no non-loop branch instructions in the program, while the upper limit is because of the fact that no node in the graph has more than two children. However, note that an upper limit of $2N$ is overly pessimistic for real programs, because (approximately) only one in every seven instructions in real-life programs are branch instructions. Hence, the time complexity of the depth-first traversal step is $O(N)$. For the program modification step, the time complexity is $O(|\mathbb{E}|)$, which because of the argument presented just now is $O(N)$. The time complexity of ranking the instructions based on the number of paths on which they lie is $O(N \log N)$, assuming an efficient sorting algorithms such as "Heapsort". Hence, the overall time-complexity of the obfuscation procedure is $O(N \log N)$.

To estimate the value of the average number of modifications made per path (M_{av}), it is essential to find the number of modifications made on every path individually, as well as the total number of paths. The total number of paths can be found during the first depth-first search. However, finding the number of modifications made individually on each path will require $O\left(\sum_{i=1}^{|\mathbb{P}|} |p_i|\right)$ steps, where $|\mathbb{P}|$ stands for the total number of paths, and $|p_i|$ is the length of the i-th path in the set of paths $\mathbb{P}$.

Space Complexity. The space complexity of the entire procedure is $O(N)$, the space required to store the information about the instructions constituting the program. If the program to be processed is of considerable size, it should be partitioned into segments of manageable sizes; each segment can be obfuscated independently and then the obfuscated segments are to be integrated to get the obfuscated program in its entirety.

4 Automation of the Obfuscation Technique

The program obfuscation methodology described in Section 2 was implemented through an automated flow, as shown in Fig. 2. The top-level tcsh script ***sobfus*** accepts as input arguments the un-obfuscated MIPS program segment in a single file (let it be "file.mips"), the number of modifications (M) to be made and the *modification radius* (r_{mod}). M is estimated a-priori from the size of the modification code blocks in the modification pool, the size of the program, and the maximum code size overhead acceptable. ***sobfus*** invokes the TCL script ***format_code*** which formats the input code by removing all comments and blank lines and replacing all labels for branch instructions in the program by

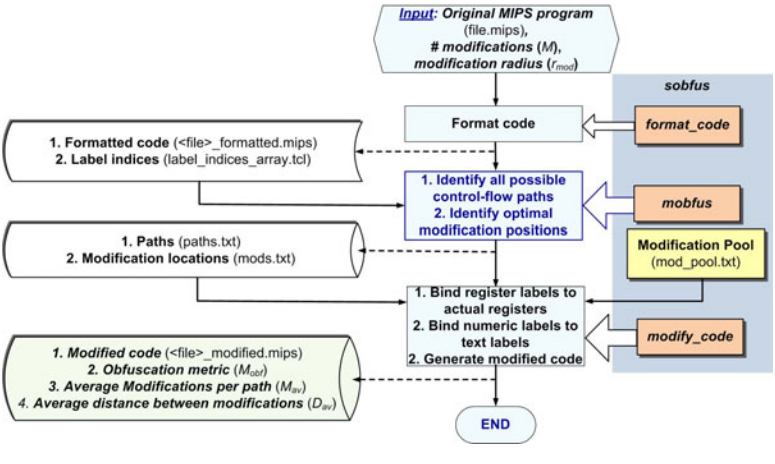

Fig. 2. Automation of the proposed obfuscation technique

Table 1. Functionality of the MIPS assembly programs used to evaluate the proposed obfuscation technique. The test programs cover a variety of representative applications from embedded domain.

Program	**Functionality**
TokenQuest.mips	One player adventure game
hanoi.mips	Recursive solution of the "Tower of Hanoi" problem
MD5.mips	MD5 hashing of a given ASCII text file
connect4.mips	Two player "Four in a Line" game
DES.mips	Digital Encryption Standard (DES) encrypter/decypter (for ASCII text files)
sudoku.mips	*Sudoku* puzzle
ID3Ediror.mips	Reading and editing of ID3 tag information in MP3 music files
string.mips	MIPS implementation of the functions of the C standard header "string.h"
cipher.txt	Various cipher techniques for ASCII text
decoder.mips	MP3 music format decoder

the corresponding destination line numbers. It produces a formatted version of the program in the file "file_formatted.mips", and a hash of the program labels and the corresponding line numbers in the file "label_indices_array.tcl". ***sobfus*** then calls the C program ***mobfus*** which enumerates all the possible control-flow paths in the program segment using Algorithm-1, and finds the optimal modification locations using Algorithm-2. It reports the enumerated paths in the file "paths.txt" and the modification locations in the file "mods.txt". ***sobfus*** then invokes the TCL script ***modify_code*** which finally produces the obfuscated program in the file "file_obfuscated.mips" by using the modification code blocks provided in the file "mod_pool.txt", and binds the register mnemonics to registers available at a given point in the program (as described in Section 2.1 and elucidated in Section 2.2). It also produces an estimate of the obfuscation metric M_{obf} according to eqn. 3, and values for the metrics M_{av} and D_{av}.

To extend the proposed obfuscation technique to binary executables, one would need to disassemble the equivalent assembly language program from a given binary, substitute all absolute addresses by symbolic addresses, apply the proposed obfuscation technique, and then again convert it back to the binary form. Note that the address substitution is essential because the insertion of modification code fragments shifts the relative positions of the instructions. Disassembly and de-compilation of binary code to assembly language code is not very difficult, and free tools are available online [33] to serve the purpose.

5 Results

The proposed technique was applied on a suite of MIPS programs varying in size from 109 to 21024 instructions. The test programs represent components of various embedded applications. The functionality of the programs are listed in Table 1. The functionality of the original and the obfuscated versions of all the programs were verified using the *SPIM* simulator [34]. The program obfuscation

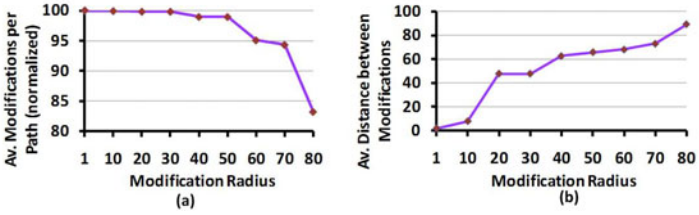

Fig. 3. Variation of (a) average modification per path (M_{av}) and (b) the average distance between modifications (D_{av}) vs. the modification radius (r_{mod}), in the program *connect4.mips*, for $M = 3$ modifications

methodology described in Section 4 was implemented and the programs were simulated on a Linux workstation with 2GB of main memory and a 2GHz quad-core processor.

We investigated the effect of variation of the *modification radius* (r_{mod}) on the average modifications per path (M_{av}) and the average distance between modifications (D_{av}) for the $N = 270$ instruction program *connect4.mips*. The number of modifications (M) was set at 3, and r_{mod} was varied between 1 and 80. Fig. 3 shows the plots of M_{av} and D_{av} vs. r_{mod}. The values for M_{av} were normalized with respect to its value at $r_{mod} = 1$ (the minimum possible value of r_{mod}). The trends are as expected, with M_{av} decreasing with r_{mod} and D_{av} increasing with r_{mod}. Note that the metrics M_{av} and D_{av} satisfy the constraints $1 < M_{av} \leq M$ and $r_{mod} \leq D_{av} < \frac{N}{M-1}$, as stated in Section 2.1.

Table 2 shows the effects of applying the proposed application technique on the MIPS program suite, at a *modification radius* ($r_{mod} = 50$), with a 10% target code-size overhead. For the largest program *decoder.mips*, only 1000 paths were considered to keep the memory requirement manageable, and r_{mod} was set to 500. As is evident from the obtained M_{obf} values, the proposed technique can provide high levels of protection at a nominal code-size overhead of 10%. Note that in larger programs and in programs with higher number of "load" and "branch" instructions, the effectiveness of the technique increases.

Table 3 shows the code-size overhead of the obfuscated program (with respect to the original program), the CPU time and average increase in execution cycles to implement algorithms 1 and 2. The run-time overheads were not calculated by direct functional simulations by SPIM, but by indirect analysis of the original and modified programs. The average increase in execution time was estimated by calculating the average increase in execution cycles per modification, and then multiplying the quantity with the average number of modifications per path. The CPU time has a strong correlation to the number of paths in the program, and a weaker correlation to the program size. These trends are consistent with the analysis of Section 3.2.

Table 2. Program obfuscation efficiency for a targeted 10% code-size overhead at a modification radius $r_{mod} = 50$

Program	Program Parameters†								Obfuscation Efficiency		
	N	C	L	$\|\mathbb{P}\|$	M	n	c	l	M_{obf}	M_{av}	D_{av}
TokenQuest.mips	109	19	14	11	2	18	3	3	1.09e-20	1.55	95.0
hanoi.mips	132	20	40	169	2	16	3	3	1.43e-19	1.91	67.0
MD5.mips	250	41	35	114	4	26	5	5	6.33e-33	3.67	65.33
connect4.mips	270	72	37	4146	4	26	5	5	1.30e-33	3.47	89.33
DES.mips	372	43	64	5241	6	34	7	9	1.54e-40	5.31	68.00
sudoku.mips	436	110	43	111113	8	41	9	11	2.66e-49	6.76	58.29
ID3Editor.mips	878	160	134	98724	12	89	16	19	1.71e-106	5.66	79.45
string.mips	876	156	224	111075	12	89	16	19	4.42e-103	10.90	60.55
cipher.mips	1956	231	218	150129	27	188	35	43	1.65e-222	26.23	75.12
decoder.mips‡	21024	174	231	1000‡	27	188	35	43	$<10^{-400}$	13.50	502.00‡

†The meaning and significance of these parameters are as described in Section 3.

‡Only 1000 paths were enumerated, and r_{mod} was set to 500.

Table 3. Overheads for the obfuscation technique (with parameters of Table 2)

Program	Overheads		
	Code-size (%)	CPU time (s)	Average Increase in Execution Cycles
TokenQuest.mips	18.85	0.10	17.83
hanoi.mips	12.12	0.40	20.06
MD5.mips	10.40	0.90	31.20
connect4.mips	9.63	1.00	29.50
DES.mips	9.14	2.00	41.60
sudoku.mips	9.40	66.00	48.17
ID3Editor.mips	10.14	112.00	54.71
string.mips	10.16	217.00	105.37
cipher.txt	10.61	1474.00	241.90
decoder.mips	0.89%	1840.00	124.50

6 Conclusions

Security of embedded software has emerged as a major challenge because of their increasing vulnerability to piracy and malicious modifications. Severe constraints on hardware and energy resources of embedded devices often limit the applicability of complex hardware and software protection approaches. We have presented a low-overhead "execution trace dependent control-flow obfuscation" technique, which requires the application of an input-dependent set of validation keys to enable a software module to function properly. The key verification mechanism is implemented by distributing the verification code throughout the program to balance the code overhead and proximity of the modifications. We have theoretically analyzed the level of security and the associated computational overhead. Application of the algorithm on a suite of MIPS programs resulted in high levels of security at nominal code size and modest computational

overhead. The technique can be easily automated and applied to arbitrarily large programs by appropriate program partitioning. Future work would involve implementation of a working prototype (including proper hardware support) of the proposed obfuscation scheme.

References

1. Turley, J.: The two percent solution, http://www.embedded.com/story/OEG20021217S0039
2. Gwennap, L., Byrne, J.: A Guide to High-Speed Embedded Processors. The Linley Group (2008)
3. Dube, R.: Hardware-based Computer Security Techniques to Defeat Hackers. ch. 5. John Wiley and Sons, Chichester (2008)
4. Ravi, S., Raghunathan, A., Kocher, P., Hattangady, S.: Security in embedded systems: design challenges. ACM Transactions on Embedded Computing Systems 3(3), 461–491 (2004)
5. Kerckhoff, A.: La cryptographie militaire. Journal des Sciences Militaires IX, 5–38 (1883)
6. Chang, H., Atallah, M.J.: Protecting software code by guards. In: Sander, T. (ed.) DRM 2001. LNCS, vol. 2320, pp. 160–175. Springer, Heidelberg (2002)
7. Barak, B.: Can we obfuscate programs?, http://www.math.ias.edu/~boaz/Papers/obf_informal.html
8. Barak, B., Goldreich, O., Impagliazzo, R., Rudich, S., Sahai, A., Vadhan, S.P., Yang, K.: On the (im)possibility of obfuscating programs. In: Conference on Advances in Cryptology (2001)
9. Collberg, C., Thomborson, C., Low, D.: Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs. In: ACM Symposium on Principles of Programming Languages (1998)
10. Collberg, C., Thomborson, C.: Watermarking, Tamper-Proofing, and Obfuscation – Tools for Software Protection. IEEE Transactions on Software Engineering 28(8), 735–746 (2002)
11. Collberg, C., Thomborson, C., Low, D.: Breaking abstractions and unstructuring data structures. In: International Conference on Computer Languages (1998)
12. Linn, C., Debray, S.: Obfuscation of executable code to improve resistance to static disassembly. In: ACM Conference on Computer and Communications Security (2003)
13. Hou, T.W., Chen, H.Y., Tsai, M.H.: Three control flow obfuscation methods for Java software. IEE Proceedings 153(2), 80–86 (2006)
14. Barak, B., et al.: On the (Im)possibility of obfuscating programs. In: Kilian, J. (ed.) CRYPTO 2001. LNCS, vol. 2139, pp. 1–18. Springer, Heidelberg (2001)
15. White, S.R., Comerford, L.: ABYSS: An architecture for software protection. IEEE Transactions on Software Engineering 16(6), 619–629 (1990)
16. Dallas Semiconductor, Dallas DS5240 Secure Microcontroller, http://datasheets.maxim-ic.com/en/ds/DS5240.pdf
17. Trusted Computing Group, Trusted Platform Module: Design Principles, http://www.trustedcomputinggroup.org/resources/tpm_main_specification
18. Trusted Computing Group, TCG Mobile Trusted Module Specification, http://www.trustedcomputinggroup.org/files/resource_files/87852F33-1D093519AD0C0F141CC6B10D/Revision_6-tcg-mobile-trusted-module-1_0.pdf

19. Leavitt Communications, Will proposed standard make mobile phones more secure?, http://www.leavcom.com/ieee_dec05.htm
20. Joepgen, H.G., Krauss, S.: Software by means of the protprog method. Elektronik 42(17), 52–56 (1993)
21. Schulman, A.: Examining the Windows AARD detection code. Dr. Dobbs Journal 18(9), 42, 448, 89 (1993)
22. Jakubowski, M.H., Saw, C.W., Venkatesan, R.: Tamper-tolerant software: Modeling and implementation. In: Takagi, T., Mambo, M. (eds.) IWSEC 2009. LNCS, vol. 5824, pp. 125–139. Springer, Heidelberg (2009)
23. Aucsmith, D.: Tamper resistant software: an implementation. In: Anderson, R. (ed.) IH 1996. LNCS, vol. 1174, pp. 317–333. Springer, Heidelberg (1996)
24. Lie, D., et al.: Architectural support for copy and tamper resistant software. ACM SIGPLAN Notices 35(11), 168–177 (2000)
25. Arora, D., Ravi, S., Raghunathan, A., Jha, N.K.: Hardware-assisted run-time monitoring for secure program execution on embedded processors. IEEE Transactions on VLSI 14(12), 1295–1308 (2006)
26. Fiskiran, A.M., Lee, R.B.: Runtime execution monitoring (REM) to detect and prevent malicious code execution. In: IEEE International Conference on Computer Design (2004)
27. Zhuang, X., Zhang, T., Lee, H.S., Pande, S.: Hardware assisted control flow obfuscation for embedded processors. In: ACM International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (2004)
28. Chakraborty, R.S., Bhunia, S.: HARPOON: An obfuscation-based SoC design methodology for hardware protection. IEEE Transactions on CAD 28(10), 1493–1502 (2009)
29. Chakraborty, R.S., Bhunia, S.: RTL hardware IP protection using key-based control and data flow obfuscation. In: VLSI Design (2010)
30. Copeland, B.J. (ed.): The Essential Turing: Seminal Writings in Computing, Logic, Philosophy, Artificial Intelligence, and Artificial Life Plus the Secrets of Enigma. Oxford University Press, Oxford (2004)
31. Dube, R.B.: Hardware-based Computer Security Techniques to Defeat Hackers. ch. 5. John Wiley and Sons, Chichester (2008)
32. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 2nd edn. ch. 22. MIT Press, Cambridge (2001)
33. The Boomerang Decompiler Project, Boomerang: A general, open source, retargetable decompiler of machine code programs, http://boomerang.sourceforge.net
34. Larus, J.: SPIM: A MIPS32 simulator, http://pages.cs.wisc.edu/~larus/spim.html
35. Balakrishnan, A., Schulze, C.: Code obfuscation literature survey, http://pages.cs.wisc.edu/~arinib/writeup.pdf
36. Patterson, D.A., Hennessy, J.L.: Computer Organization and Design: The Hardware/Software Interface (Appendix A), 4th edn. Morgan Kaufmann Publishers, San Francisco (2009)