A Security Model and Implementation of Embedded Software Based on Code Obfuscation

1st Jiajia Yi
University of Electronic Science and Technology of China
Chengdu, China
yijiajia@std.uestc.edu.cn

2rd Lirong Chen
University of Electronic Science and Technology of China
Chengdu, China
lrchen@uestc.edu.cn

3nd Haitao Zhang
University of Electronic Science and Technology of China
Shanghai Automotive Industry Corporation Technology Center
Shanghai, China
zhanghaitao01@saicmotor.com

4th Yun Li
Guangdong Weichen Information Technology Co., Ltd
Guangdong, China
liy@vecentek.com

5th Huanyu Zhao
Guangdong Weichen Information Technology Co., Ltd
Guangdong, China
zhaohy@vecentek.com

***Abstract*—Current approaches for the security of embedded software mainly focused on some specific platforms. In this paper, a security model based on code obfuscation is applied to embedded software. A control flow flattening algorithm is used to implement an automated obfuscator, which obfuscates C code first, and does source-to-source conversions to protect software on different platforms. The effectiveness of code obfuscation is evaluated by a multi-level quantitative model proposed in this paper. Related experiments are carried out on the NUC140VE3CN board and MC9S12XEP100MAG board, which are typical hardware platforms used in the application domain of automotive. The result of experiments shows that for one thing, the quantitative value of the effectiveness of the obfuscated program is obviously higher than that of the original program, namely the strength for software to keep it from being reversed is greater, and the overhead of time and space is acceptable; for another, the efficiency of the evaluation model is also demonstrated.**

***Keywords*— software security, code obfuscation, embedded system, evaluation model**

I. Introduction

Intelligent and connected embedded systems have not only brought convenience to life, but also introduced potential security threats. For example, the automotive industry, a typical application domain of embedded systems, is under the threat of illegal intrusion. Jeep and Tesla, known by the community as reported, were hacked into their internal control systems. With the emergence of serious problems, security is no longer a negligible factor in embedded system design [1]. While it is not that easy to protect embedded systems because of their limited resources and high demand for real-time performance. The security mechanism of PC usually not suitable for embedded systems.

There are many embedded devices with extremely limited software and hardware resources, especially for deep embedded devices such as engine control units and vehicle control systems. Therefore, a suitable scheme starting from the security of embedded software itself to improve the security of the system is needed. By carefully handling sensitive information, the security of embedded software can be improved significantly and thus can improve the whole. There are some commonly used software protection technologies in embedded systems such as watermarking, tamper-proofing, obfuscation [4], secure boot [5], control flow integrity check [5], TrustZone technology [6], fingerprint and encryption [8], address space layout randomization [9], authentication [10], secure processor [11], memory protect [12], the secrecy capacity of wiretap channel [13], etc. The main idea of code obfuscation is to transform the code into a form that is functionally equivalent but difficult to read and understand by illegal hackers, making it more difficult for the reverse engineering [2]. Compared with other protection technologies mentioned above, code obfuscation combines the following advantages:

1) Significant protection effect: The equivocal program logic after obfuscation increases the difficulty of malicious analysis and the cost of reverse engineering.

2) Wide application prospects: Either source code or intermediate code can be obfuscated, which are more fixed in format and widely applicable to different environments than binary code. We do not disguise the fact that it has been applied to our automobile electronic software protection on different platforms.

3) Technical feasibility: Neither additional hardware knowledge nor any specific research on embedded hardware architecture is needed, we just focus on the characteristics of the source code itself.

4) Low overhead: The low overhead of code obfuscation determines that it is appropriate for the systems with limited resources.

Although obfuscation technology plays an important role in software security, the current researches in the field of embedded system are mainly limited to the specific architecture, hardware obfuscation, or the operation of some instructions. Our contributions are as follows:

- Comprehensively considering the characteristics and special requirements of embedded systems, a security model for embedded software based on code obfuscation is proposed.
- A multi-level quantitative model is proposed to evaluated the effectiveness of code obfuscation.
- An automatic obfuscator used to protect software on multiple embedded platforms has been implemented.
- The designed experiments verify the function of the obfuscator and the correctness of the evaluation model.

The rest of this paper is organized as follows. Section II discusses the background and the threat model. Section III discusses some feasible code obfuscation methods. Section IV is a detailed introduction of the multi-level quantitative evaluation model. Section V elaborates on the design and implementation of the obfuscator. Section VI reports the evaluation results of the proposed method. Section VII gives the conclusion.

II. Background

A. Security threats to embedded systems

Software attacks are major threats to embedded systems, especially for those can download code remotely. These attacks may be implemented by malicious agents such as viruses, worms, Trojans. They can weak the security of embedded systems on various aspects such as integrity, confidentiality and availability. Malicious software agents craftily use the vulnerabilities or exposures of system architecture to launch software attacks. Compared with traditional PC computer systems, embedded systems usually have strict constraints on CPU speed, memory capacity, power consumption, etc. The security challenges of embedded systems include the following aspects:

1) In terms of resources : The resources, cost and volume of embedded systems are extremely limited, whereas higher computing power and real-time performance are required, which is significantly different from traditional (or general purpose) computer systems. Security strategies for traditional computer systems can not be simply applied to embedded systems.

2) In terms of working environment: Embedded systems work in different environments and may be exposed to unsafe or untrusted cases, which increases the probability of being threatened and attacked.

3) In terms of software vulnerabilities: Embedded applications, databases and operating systems are huge in scale, complex in structure and function. Problems such as data stealing, code usurping that may occur on each key point seriously threaten the security of the system.

4) In terms of network attack: Nowadays most embedded systems are connected to the Internet. Hackers can remotely control the systems or steal sensitive information via the network connections.

B. Threat model

We assume that the attackers have taken control of the program. Malicious reverse analysts illegally access memory or obtain internal information of the software. They will attack the system, tamper or copy code, or take other illegal actions. There are some technologies to protect embedded system to some extent, such as random address space layout, memory permissions check, authentication, control flow integrity check, and schemes like TEE (Trusted Execution Environment) or TrustZone. But they are unable to guarantee absolute security, especially for those whose resources are too limited to support these technologies . Attackers may read the executable code from memory statically, or obtain the control logic of the code by dynamically monitoring the execution process. Their next step is to reverse the program.

The main goal is to increase the difficulty and the cost of reverse analysis, especially for the embedded systems which are lack of physical protection and easy to disclose their code. Code obfuscation is applied to complicate the logic of the program and reduce their readability to improve the software security of embedded systems.

C. Related work

New methods and algorithms are putting forward constantly in the research of code obfuscation, but more emphasis is put on the exploration of obfuscation technology itself and seldom combined with the features and special requirements of embedded systems. Collberg et al. [2] defined and classified the obfuscation technology, and put forward the evaluation indicators. Chenxi Wang proposed a control flow flattening algorithm and proved the effectiveness against static analysis [14], [15]. The realization of control flow flattening algorithm of C++ was proposed in [16]. D. Xu et al. [17] sliced the code and generated multiple random execution paths. Xie et al. [18] reduced the accuracy of disassembly by overlapping instructions and jumping instructions. Behera et al. tampered the code and embedded self-modifying instructions to disrupt static disassembly [19]. B. Yadegari [20] securely obfuscated the class of conjunction functions. A approach that used semantically equivalent to code clone within the source code to obfuscate logical part of program text for JAVA [21]. However, none of the above have addressed the features and requirements of embedded systems, nor have taken the multiple running environment of the embedded software into consideration. The previous achievements serve as the basis of our work, on which we have made some improvements to make them more suitable for embedded scenarios.

There are also some researches about software security for embedded systems, but they mainly focused either on hardware obfuscation, or on a specific architecture. LOCO [22] realized a code obfuscation tool for some different platforms. But it gradually does not work well on the new embedded devices that appear over time. Related researches also include code obfuscation on Android embedded devices [23], LLVM code obfuscation under Win32 [24], code obfuscation based on specific instructions swapping [25], hardware obfuscation at the microarchitecture level on FPGA [26], the key-based control flow obfuscation scheme for MIPS assembly programs [27], the cost-effective obfuscation techniques based on instruction set randomization on FPGA [28], etc. However, these studies only implemented the security technologies for specific platforms with limited scenarios. A more general and automated code obfuscation tool works on different platforms is needed.

The most appropriate evaluation methods for their researches were also applied. For example, Tímea László [16] used McCabe's cyclomatic complexity metric [3] directly, which is common in the research of obfuscation technology. While the methods of analysis of the dynamic CFGs [25], [28], the changed distribution of instruction [26] as well as the strict overhead constraints were considered in the embedded software evaluation scheme [28]. Our obfuscator is neither a pure conversion of C language, nor is dependent on instruction and architecture. Our evaluation model combines the two to support the points discussed in this paper.

III. Feasible code obfuscation methods

Code obfuscation hide the important information of the program without changing the functionalities of the original software. It can protect the storage of key data, sensitive information, core procedures or algorithms, the calling processes between programs or functions, the loopholes contained in the code, etc. Increasing the complexity and fuzziness of the program will reduce the readability of the program and increase the difficulty of automatic reverse analysis and manual static analysis. The cost of reverse engineering is far greater than the benefit, which is the key idea to prevent software from being attacked by malicious reverse engineering. The definition of code obfuscation is described as below [2]:

We record P as the original program, P' as the target program, and T as the transformation from P to P'. That is, $T: P \rightarrow P'$. If the original program P and the transformed target program $P' = T(P)$ are functionally equivalent with different forms, then T is called the obfuscation transformation.

There are many types of code obfuscation, but we have to find one that is most suitable for embedded systems. Code obfuscation can be categorized into static and dynamic obfuscation. The typical representatives of dynamic obfuscation are self-modification code technology (SMC), virtual machine protection technology (VMP), etc. This kind of obfuscation requires dynamic changes during the execution. Dynamic obfuscation technology often brings a lot of cost, which is not suitable for embedded systems with the requirements of high real-time and the limitation of resources. The range of static obfuscation objects is very wide, including source code, intermediate code and binary executable program. And changing the shape of the program, protecting and hiding the data, flattening the control flow, adding some opaque predicates to change the structure of the program are all static obfuscation technologies. Static obfuscation means that the program has been modified before execution, and program will not change dynamically, nor additional monitoring is required. Therefore, static obfuscation brings less cost and is more suitable for embedded systems.

Shape obfuscation, data obfuscation and control flow obfuscation are all well know and classic static obfuscation algorithms. There are three reasons that control flow obfuscation is feasible to protect embedded system software.

(1) Shape obfuscation changes the key information of the program. But the identifiers, function names, variable names and other information in the source code can be optimized by the compiler instead of being kept in the binary executable file in C language, the widely used language in embedded systems. Therefore, it is of little meaning to design a C language obfuscator with shape obfuscation. (2) The purpose of data obfuscation is to prevent attackers from extracting important data from programs. In the software of embedded systems, encryption is often used to protect the data instead of the software code. And (3) the control flow obfuscation improves the security of software by changing the structure of source code and complicating the logic of the program. The method is difficult to be anti-obfuscated or analyzed statically by manual or automatic disassembly tools. Therefore, control flow flattening algorithm is used to design and implement a source code obfuscation tool.

The program's control flow is "flattened" after control flow flattening obfuscation as shown in figure 1 and figure 2. The blocks originally belonging to different levels are placed at the same level, which makes the static analysis a more complex work because of its chaotic loop.

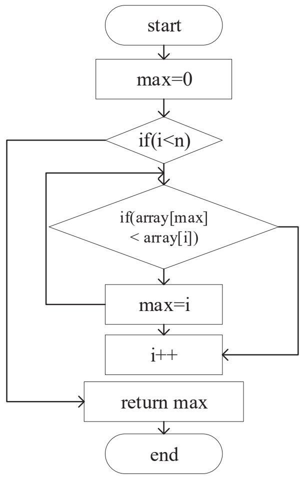

Fig. 1. The control flow of the original program.

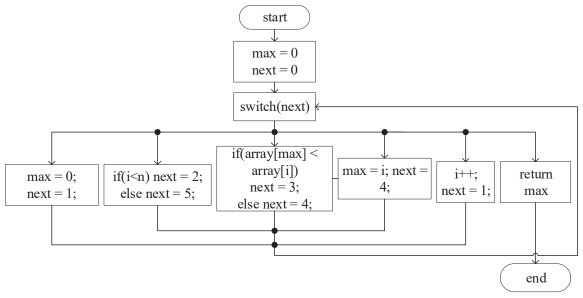

Fig. 2. The control flow after obfuscation.

IV. A MULTI-LEVEL QUANTITATIVE MODEL

Software evaluation indicators were proposed by Colberg [2]: potency, resilience and cost, which respectively represent the complexity of obfuscation algorithm, the resistance of the obfuscated program to reverse analysis, and the extra time and space overhead caused by code obfuscation. In addition, embedded software usually has strict requirements for real-time performance, storage space and other hardware resource costs. Therefore, a multi-level quantitative model is proposed to guide the design and evaluate the performance for our obfuscator in the context of embedded system. It is an intuitive representation of the requirements for embedded software scenarios. We propose the concept of **quantitative value of effectiveness** for code obfuscation, which presents: the quantification of security, time overhead and space overhead after obfuscation under the premise of ensuring all functionalities unchanged.

A. Model building

The evaluation model consists of four layers: target layer (V_1), middle layer 1 (V_2), middle layer 2 (V_3), factor layer (V_4). The weight of each layer is W_1, W_2, W_3, W_4 respectively. These multi-level weighted attributes are used to establish the mathematical model for quantitative evaluation. Figure 3 shows the hierarchical structure of the evaluation model.

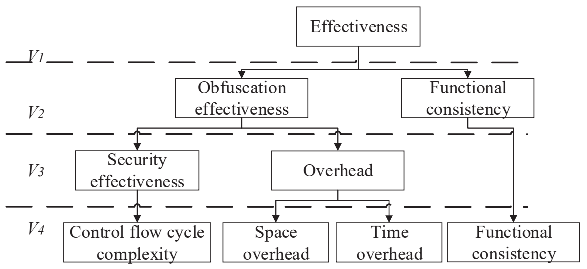

Fig. 3. Hierarchical structure of multi-level quantitative model.

The mathematical definition of the model is as follows:

$$Target: F = \begin{cases} 0 & , \exists v_{2j} = 0 \\ \vec{V_1} \bullet \vec{W_1}, \forall v_{2j} = 1 \end{cases} \tag{1}$$

Where F is the quantitative result of effectiveness, W is the weight of elements in the corresponding level, and v_{ij} is the element of layer V_i. F is equal to 0 when the functionalities of some codes changed after obfuscation in V_4 . The following describes the specific steps of establishing the model.

First we get the input matrix *input*. There are k pieces of codes or projects in the test set. We test their n attributes as the elements of *input* in V_4, and the *input* matrix has the forms:

$$input = \begin{bmatrix} attr_1^1 & attr_2^1 & attr_3^1 & \cdots & attr_n^1 \\ attr_1^2 & attr_2^2 & attr_3^2 & \cdots & attr_n^2 \\ attr_1^3 & attr_2^3 & attr_3^3 & \cdots & attr_n^3 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ attr_1^k & attr_2^k & attr_3^k & \cdots & attr_n^k \end{bmatrix} \tag{2}$$

Where $attr^i_j$ represents the *j-th* attribute from the *i-th* code. In our model, the value of attribute includes the increase percentage of control flow cyclomatic complexity, the reciprocal of the increase percentage of space overhead, the reciprocal of the increase percentage of time overhead and the judgment of functional consistency. We hope that the smaller the overhead of program growth after obfuscation is, the better. So we take the percentage of space overhead and time overhead as the reciprocal to reduce the quantitative value. It is specified that $attr^i_n$ refers to "*functional consistency*", shown in formula (3).

$$attr_n^i = \begin{cases} 0 & , functional \quad consistency \\ 1 & , functional \quad inconsistency \end{cases} \tag{3}$$

We get $attr^i_n = 0$, when functionalities of some codes changed after obfuscation. Then $F = 0$ and the evaluation ends. Otherwise we proceed as follows.

First, normalize *input* matrix (4):

$$b_{ij} = \frac{attr_j^i}{\max(attr_j^1, attr_j^2, \cdots, attr_j^k)} \quad (i = 1, 2, \cdots, k) \tag{4}$$

And the normalized input matrix B as the output of V_4.

$$B = \begin{bmatrix} b_{11} & b_{12} & b_{13} & \cdots & b_{1(n-1)} \\ b_{21} & b_{22} & b_{23} & \cdots & b_{2(n-1)} \\ b_{31} & b_{32} & b_{33} & \cdots & b_{3(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ b_{k1} & b_{k2} & b_{k3} & \cdots & b_{k(n-1)} \end{bmatrix} \tag{5}$$

Second, the weight of each layer is determined. In different demanding background, the weight is not always the same. In order to simplify the model, the following methods shown in formula (6) are used to determine the weight of each layer: V_1 and V_2 directly describe the result of obfuscation, so their weights are set to 1 directly; we can put the weight of security effectiveness w_3^1 and overhead w_3^2 more flexibly according to the needs in actual situation because the overhead of security and security effectiveness in embedded system is usually positively related in V_3.

$$\begin{cases} w_1 = w_2 = 1 \\ w_3^1 = \alpha, w_3^2 = 1 - \alpha \end{cases} \tag{6}$$

Formula (7) shows the method to calculate effectiveness and overhead in V_3. The input of V_3 is normalized as matrix B and its weight is W_3.

$$\begin{cases} V_{31} = b_{11}w_{31} + b_{12}w_{32} + \cdots + b_{1(n-1)}w_{3(n-1)} \\ V_{32} = b_{21}w_{31} + b_{22}w_{32} + \cdots + b_{2(n-1)}w_{3(n-1)} \\ \cdots \\ V_{3k} = b_{k1}w_{31} + b_{k2}w_{32} + \cdots + b_{k(n-1)}w_{3(n-1)} \end{cases} \tag{7}$$

Combining the calculation of the V_1 and V_2, we get F^k in formula (8), the **quantitative value of effectiveness** of obfuscation when the functionality of *k-th* code or project is consistent.

$$\begin{cases} F^1 = (\sum_i V_{31}^i w_{2i}) w_1 \\ F^2 = (\sum_i V_{32}^i w_{2i}) w_1 \\ \cdots \\ F^k = (\sum_i V_{3k}^i w_{2i}) w_1 \end{cases} \tag{8}$$

V. Software Implementation

A. Challenge

Our automatic code obfuscator based on control flow flattening combines the characteristics and functional requirements of embedded system, and takes the balance between the security and overhead into consideration. There are four challenges to implement such an obfuscator: the program before and after obfuscation must be functionally equivalent , the obfuscator can be applied in different embedded platforms, C language can be obfuscated by control flow flattening algorithm, and balancing the security and cost.

1) Functionally equivalent : The syntax and semantics of the program is needed to achieve this basic requirement, and then equivalent transformation will be carried out. With the help of compiler, we obtain abstract syntax tree (AST) of the program and replace the original nodes in the tree with equivalent content to rewrite the code into C code. However, embedded software often needs specific non-open-source compilers, so it is unlikely to get abstract syntax tree of different programs from those compilers. Our approach is to rewrite the open source C compiler and our choise is Clang.

2) Being available for different platforms: Embedded platforms run binary code compiled from high-level program language especially C language, but there are many different hardware platforms running different binary code and using different instruction sets. Our design is a general-purposed obfuscator which can be used in different platforms. The obfuscator works in source level conversion. It can automatically obfuscate C code with control flow flattening algorithm, and then output C code. The AST is obtained and the program is transformed before generating the assembly code. This step is independent of instruction set and architecture.

3) Fully considering characteristics of C language: C language has some grammar rules, such as the requirements of variable declaration, assignment and use, the scope of function and variable, the execution order of branch structure and so on. Therefore, when using the control flow flattening algorithm to obfuscate programs written in C language, we should pay attention to these rules to ensure the correct execution of the program after obfuscation.

4) Balancing the security and cost: Improving software security will bring extra overhead. In order to reduce it, we firstly chose a appropriate obfuscation algorithm. Then we noticed that only the critical parts of a project needs to be protected instead of all. Therefore, the obfuscator performs obfuscation in units of functions, and the user decides which functions need to be obfuscated.

B. System framework

The framework of the system is shown in Figure 4. *P* is the input program, namely the original target program, and the output program after obfuscation is *P*'. *P* and *P'* are required to be functionally consistent. After being preprocessed and compiled, the abstract syntax tree and control flow graph of *P* are obtained for further obfuscation. Obfuscation algorithm ensures that *P'* is more difficult to be reversed.

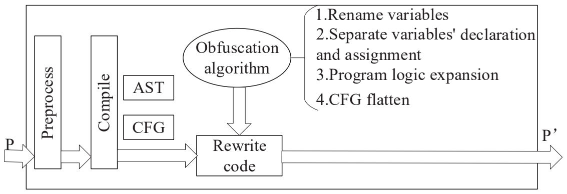

Fig. 4. System framework.

C. Algorithm implementation

The obfuscator works on function basis so that users can choose functions those are critical to protect in order to reduce the cost. Our obfuscator modifies the abstract syntax tree and rewrites the modified abstract syntax tree into C program to realize the conversion from source to source. The nodes of the abstract syntax tree contain statements, declarations and expressions of functions. In the process of obfuscation, those messages of each function are firstly obtained from the abstract syntax tree, and then they are respectively obfuscated by control flow flattening algorithm. The basic idea of control flow flattening algorithm is to break and blur the relationship between the original blocks in the program [16]. It uses *switch-case* to replace *for, while, do-while* and other statements in the program. The basic method is as follows.

Firstly, the control flow graph is constructed and basic blocks of each function are acquired. Secondly, each basic block is encapsulated in a different *case*, and all *cases* are encapsulated in a *switch*. Then a *dispatch* variable is used to determine the execution order of the basic blocks. Finally, the *switch-case* statement is encapsulated in a *loop* body. However, there are several problems need to be solved because of the characteristics of C language.

1) Declaration and use of variables: In C language, variables need to be declared at first and then used. But the declaration and use of local variables may be divided into different branches as the control flow changes, and errors may occur. We bring forward the declaration of all variables to the beginning of the function to solve this problem,.

2) Variable scope and namespace problem: Local variables are allowed to use the same name in different scopes. But advancing the declaration of all variables may cause conflicts because of the variables with a same name. In addition, we also need to solve the problem that the declaration and assignment of const variables cannot be separated, the processing problems of extern variables, the processing problems of reference types and so on.

3) Execution sequence: The change of control flow may cause the problem of execution sequence and the functionalities of the program may be changed. We must make sure that the program has a correct execution sequence.

The following steps are taken to implement automatic source code level obfuscator. The original program to be obfuscated is shown in Figure 5, for example. Function *fun* includes *variables*, *do-while* statements, *if* statements, and some operations.

```
void fun(){
   int turn = 1;
   do{
      turn ++;
      printf("turn = %d",turn );
      if(turn == 15){
         int newnum = turn * 2;
         printf("newnum= %d",newnum);
      }
   }while(turn <20);
}
```

Fig. 5. Program before obfuscation.

1) Preprocessing: The program is preprocessed and compiled to obtain valuable information such as control flow graph and abstract syntax tree.

2) Renaming variables to solve conflict on variables of the same name: Develop a renaming mechanism to name all variables as globally unique names. It is worth noting that global variables may be referenced by other programs and global variables have unique names, so they cannot and do not need to be renamed. And external variables do not need to be renamed, typedef does not need to be renamed, enumeration type do not need to be renamed, structures and members can not be renamed, and other special variables cannot be renamed.

3) Advancing the declaration of variables and separating declaration from assignment: Although putting the declaration of variables in advance may change the scope and living space of variables, there will be no problem with conflict of variables because variables have already been set globally unique names in the previous step. It should be noted that the declaration and assignment of the variables decorated by const cannot be separated, the assignment of arrays need to be expanded, and the conversion of reference type needs to be pointer, etc.

4) Logical expansion : Logic extension is realized by changing *for*, *while* and *do-while* statements into *if-goto* statements. The *if-goto* statement visually indicates the content of each basic block and determines the execution order between different basic blocks. The use of variables is not a hinder regardless of which base block they are placed in after the first two steps.

5) Control flow flattening: Each basic block in the control flow graph represents a jump branch and different branches are encapsulated in the same layer, which makes the control flow of the whole program flat. As shown in Figure 6, the six basic blocks of the original program are converted into six *case* statements, and a dispatch variable (*var_2* in figure 6) is added to ensure the execution order is consistent with the original.

```
void fun(){
   int var_0,  var_1,  var_2 = 6;
   while (var_2 != 0)
      switch (var_2) {
        case 1:var_2 = 0;break;
        case 2:var_2 = 5;break;
        case 3:
            if (var_0 < 20) {var_2 = 2;}
             else {var_2 = 1;}
            break;
        case 4:
            var_2 = 3;
            var_1 = var_0 * 2;
              printf("newnum= %d",var_1);
            break;
        case 5:
            var_0++;
             printf("turn = %d",var_0 );
            if (var_0 == 15) {var_2 = 4;}
              else {var_2 = 3;}
            break;
        case 6:
            var_2 = 5;
            var_0 = 1;
            break;
      }
  }
```

Fig. 6. Program after obfuscation.

6) Rewrite: The abstract syntax tree is modified by replacing the subtree of the abstract syntax tree with the corresponding processed content after obfuscation is completed, as shown in Figure 7.

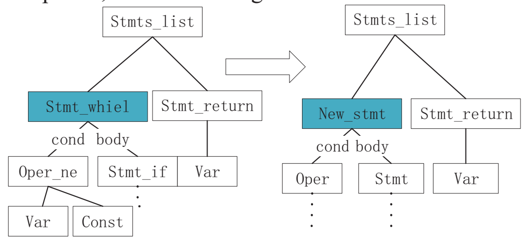

Fig. 7. Rewrite abstract syntax tree.

Rewriting the abstract syntax tree to C program, the whole code obfuscation process is completed.

VI. EXPERIMENT

A. Experimental environment

Embedded systems have different hardware and software resources according to different application environments. We conducted experiments in vehicle control equipments, whose hardware and software resources are relatively limited. They are used for engine control systems and so on, running lightweight operating systems. Experiments were carried out on two representative automotive platforms, NUC140VE3CN board and MC9S12XEP100MAG board. Parameters of the platform are shown in Table 1.

TABLE I. EXPERIMENTAL ENVIRONMENT

Platform	Parameter	Value
NUC140VE3C N	OS	μC/OS
	APROM	128KB
	RAM	16KB
	Flash	128K
	Processor	ARM Cortex M0
MC9S12XEP1 00MAG	OS	OSEK
	EEPROM	4K
	RAM	64K
	Flash	1M
	Processor	HCS12X
	Frequency	6MHz

B. Resulsts

We carefully selected some important codes in each project for obfuscation. Some codes of OSEK operating system were obfuscated on MC9S12XEP100MAG, including osekAlarm.c, osekEvent.c, osekCounter.c, osekDebug.c, osekEvent.c, osekTask.c, osekExecution.c, osekInterrupt.c, osekMessage.c, and the rest remained unchanged. We designed 5 test programs (P1 to P5) ran on OSEK operating system before and after obfuscation. And these test programs themselves were not obfuscated. The obfuscated objects on NUC140VE3CN board were all drivers and user code (P6 to P9) instead of OS code, and these programs ran on the μC/OS operating system.

1) The effectiveness of obfuscation evaluation: The strength based measurement method [3] proposed by McCabe is used to evaluate the security, which means to calculate the cyclomatic complexity of control flow. The higher the value, the greater the strength and the securer the program. Table 2 shows the values of control flow cyclomatic complexity of the tests. It can be seen that the strength of the program increased after obfuscation, which proves that obfuscation is significant.

TABLE II. CONTROL FLOW CYCLOMATIC COMPLEXITY OF PROJECTS

No.	Project	Control flow cyclomatic complexity		times
		Before	After	
P1	osekTask	607	994	1.6378
P2	osekIntCounterAlarm	481	868	1.8046
P3	osekResource	487	874	1.7947

P4	osekEvent	663	1150	1.7345
P5	osekMessage	603	990	1.6418
P6	ucosTaksManager	195	438	2.2462
P7	ucosPhilosopher_Repast	198	501	2.5303
P8	ucosTimer_Interrupter	234	498	2.1282
P9	ucosLCD_graphic	257	648	2.5214

2) Space and time overhead analysis: We tested the storage occupied by programs as shown in Table 3. And table 4 shows the running time of different projects on OSEK or μC/OS before and after obfuscation. Both of them are higher than that of the original and vary with the tests and obfuscated objects. This is related to comprehensive factors such as code size, function content, and system calls. But the scale of the increase is modest.

TABLE III. PROGRAM SIZE AND SPACE OVERHEAD

No.	**Project**	**Space (KB)**		**Times**
		Before	**After**	
P1	osekTask	680.1600	697.7060	1.0258
P2	osekIntCounterAlarm	758.6550	763.3130	1.0061
P3	osekResource	787.5730	796.9530	1.0119
P4	osekEvent	769.2570	774.6410	1.0070
P5	osekMessage	768.0130	776.7610	1.0114
P6	ucosTaksManager	388.6000	393.3760	1.0123
P7	ucosPhilosopher_Repast	394.2520	401.4080	1.0182
P8	ucosTimer_Interrupter	394.8080	398.5080	1.0094
P9	ucosLCD_graphic	354.816	360.424	1.0158

TABLE IV. PROGRAM RUNNING TIME AND TIME OVERHEAD

No.	**Project**	**Running time (ms)**		**Times**
		Before	**After**	
P1	osekTask	5.0938	5.0977	1.0008
P2	osekIntCounterAlarm	7.6092	7.6145	1.0007
P3	osekResource	8.5840	10.4262	1.2144
P4	osekEvent	8.2497	8.6560	1.0493
P5	osekMessage	9.6555	9.6597	1.0004
P6	ucosTaksManager	2.1677	4.3065	1.9867
P7	ucosPhilosopher_Repast	1.7606	3.2549	1.8488
P8	ucosTimer_Interrupter	1.3564	1.8566	1.3689
P9	ucosLCD_graphic	4.4334	8.7661	1.9772

3) Quantitative value of effectiveness evaluation: The experiment shows that functionalities of programs are equivalent before and after obfuscation, so the **quantitative value of effectiveness** is not equal to 0 according to the multi-level quantitative model proposed above as shown is table 5. Since the elements of the *input* matrix before obfuscation are all equal to 1 according to the model, the quantitative values of the original are the same (0.6976).

TABLE V. QUANTITATIVE VALUE OF EFFECTIVENESS

No.	**Project**	**Quantitative value of effectiveness**		**Times**
		Before	**After**	
P1	osekTask	0.6976	0.8221	1.1785
P2	osekIntCounterAlarm	0.6976	0.8549	1.2255
P3	osekResource	0.6976	0.8076	1.1577
P4	osekEvent	0.6976	0.8293	1.1888
P5	osekMessage	0.6976	0.8215	1.1776
P6	ucosTaksManager	0.6976	0.8166	1.1634
P7	ucosPhilosopher_Repast	0.6976	0.8808	1.2626
P8	ucosTimer_Interrupter	0.6976	0.8508	1.2196
P9	ucosLCD_graphic	0.6976	0.8708	1.2483

The average quantitative value is 0.8393 after obfuscation, which is greater than 0.6976 of the source code. The results prove that programs after obfuscation are securer and the time and space overhead is acceptable. Figure 8 shows the increased overhead of each test after obfuscation.

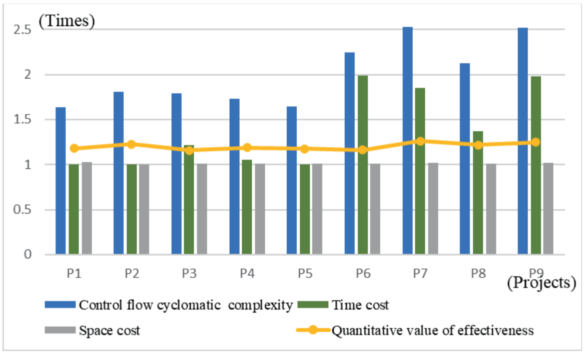

Fig. 8. The increased overhead after obfuscation.

The results also show that the more programs are obfuscated in a project, the more effective they are to resist to reverse analysis, while the higher the overhead is.

VII. CONCLUSION

We found that code obfuscation can significantly improve the security of embedded software and we have implemented an automatic code obfuscator. The obfuscator based on a control flow flattening algorithm and is designed to protect software on different platforms. We reduced the overhead of the obfuscator by choosing an appropriate obfuscation algorithm and protecting only some critical functions. A multi-level quantitative model was put forward to evaluate the effectiveness. Experiments on two different platforms verify the correctness of the model and the effectiveness of the obfuscator.

One of the novel ideas we explored was the source-level obfuscation in order to fit different embedded platforms. However, more algorithms need to be implemented to provide users with more choices and verify which method is better for embedded software. Our future work will focus on more implementation and evaluation, and then perform some optimization of the algorithm.

ACKNOWLEDGMENT

We thank the reviewers for helpful comments. This work is supported by the Major Science and Technology Special Project of Sichuan Province (No. 2018GZDZX0009) and the Introducing Program of Dongguan for Leading Talents in Innovation and Entrepreneur (Dongren Han [2018], No. 738).

REFERENCES

[1] 2017 embedded markets study. https://m.eet.com/media/1246048/2017-embedded-market-study.pdf. Accessed: 2017-5-4.

[2] 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.

[3] McCabe, T.J., Watson, A.H. “Software complexity,” in *Journal of Defense Software Engineering*, vol. 7, pp. 5–9, 1994.

[4] Christian S. Collberg and Clark Thomborson. “Watermarking, Tamper-Proofing, and Obfuscation -- Tools for Software Protection,”in *IEEE Transactions on Software Engineering*, vol. 28, no. 8, pp.735-746, 2002.

[5] Field, Scott A. , and J. D. Schwartz . “Secure boot,” European Patent EP1872231A2, January 02, 2008.

[6] Pike, Lee, Hickey, Patrick Christopher et al., “Software security via control flow integrity checking,” U.S. Patent 9846717, December 19, 2017.

[7] D. Ji, Q. Zhang, S. Zhao, Z. Shi and Y. Guan, “MicroTEE: designing TEE OS based on the microkernel architecture,” in *2019 18th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/13th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)* , 2019, pp. 26-33.

[8] M.A. Murillo-Escobar, C. Cruz-Hernández, F. Abundiz-Pérez, and R.M. López-Gutiérrez. “A robust embedded biometric authentication system based on fingerprint and chaotic encryption,” in *Expert Systems with Application*, vol. 42, no.21, pp. 8198-8211, 2015.

[9] Kevin Z. Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi, “Just-in-time code reuse: on the effectiveness of fine-grained address space layout randomization,” in *IEEE Symposium on Security and Privacy (SP)*. IEEE, 2013, pp. 574–588.

[10] W. Luo, Y. Hu, H. Jiang and J. Wang, “Authentication by Encrypted Negative Password,” in *IEEE Transactions on Information Forensics and Security*, vol. 14, no. 1, pp. 114-128, 2019.

[11] L. Ren, C. W. Fletcher, A. Kwon, M. van Dijk and S. Devadas, “Design and Implementation of the Ascend Secure Processor,” in *IEEE Transactions on Dependable and Secure Computing*, vol. 16, no. 2, pp. 204-216, 2019.

[12] Z. Liu, H. Seo, A. Castiglione, K. R. Choo and H. Kim, “Memory-Efficient Implementation of Elliptic Curve Cryptography for the Internet-of-Things,” in *IEEE Transactions on Dependable and Secure Computing*, vol. 16, no. 3, pp. 521-529, 2019.

[13] B. Dai and Y. Luo, “An Improved Feedback Coding Scheme for the Wire-Tap Channel,” in *IEEE Transactions on Information Forensics and Security*, vol. 14, no. 1, pp. 262-271, 2019.

[14] C. X. Wang. “A security architecture for survivability mechanisms,” Department of Computer Science, The University of Virginia, USA, Ph.D. Dissertation, 2000.

[15] Chenxi Wang, Jonathan Hill, John Knight, and Jack Davidson, “Software tamper resistance: obstructing static analysis of programs,” Department of Computer Science, University of Virginia, USA, Tech. Rep., 2000.

[16] Tímea László and Ákos Kiss, “Obfuscating C++ programs via control flow flattening,” in *Annales Universitatis Scientiarum Budapestinensis de Rolando Eötvös Nominatae. Sectio Computatorica*, vol. 30, pp. 2-19, 2009.

[17] D. Xu, J. Ming, and D. Wu, “Generalized dynamic opaque predicates: A new control flow obfuscation method,” in *ISC 2016, ser. Lecture Notes in Computer Science*, vol. 9866, pp. 323–342. Springer, 2016.

[18] Xin Xie, Fenlin Liu, Bin Lu, and Fei Xiang, “Mixed obfuscation of overlapping instruction and self-modify code based on hyper-chaotic opaque predicates,” in *2014 Tenth International Conference on Computational Intelligence and Security*. IEEE, 2014, pp. 524–528.

[19] Behera C K , Bhaskari D L , “Self-modifying code: a provable technique for enhancing program obfuscation,” in *International journal of secure software engineering*, vol. 8, no. 3, 2017, pp. 24-41.

[20] B. Yadegari and S. Debray, “Symbolic execution of obfuscated code,” in *22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 732–744.

[21] Pratiksha Gautam and Hemraj Saini. “A Novel Software Protection Approach for Code Obfuscation to Enhance Software Security,” in *International Journal of Mobile Computing and Multimedia Communications,* vol. 8, no. 1, 2017, pp. 34–47.

[22] Matias Madou, Ludo Van Put, and Koen De Bosschere, “LOCO: an interactive code (de)obfuscation tool,” in *proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation (PEPM '06)*. Association for Computing Machinery, 2006, pp. 140–144.

[23] Xuerui Pan, Research on static obfuscation and dynamic defense on android platform, Nanjing University, Nanjing, 2015.

[24] Zhang Quan, Shu Hui, Li Jingrui, “Research on code obfuscation based on LLVM under Win32 platform,” in *Journal of Information Engineering University*, vol. 19, no. 4, pp. 498-502, 2018.

[25] Pan Yan, Zhu Yuefei and Lin Wei, “Code obfuscation based on instructions swapping,” *Journal of Software*, vol. 30, no. 6, pp. 1778-1792, 2019.

[26] M. Fyrbiak, S. Rokicki, N. Bissantz, R. Tessier and C. Paar, "Hybrid Obfuscation to Protect Against Disclosure Attacks on Embedded Microprocessors," in *IEEE Transactions on Computers*, vol. 67, no. 3, pp. 307-321, 1 March 2018.

[27] Chakraborty R.S., Narasimhan S., Bhunia S, “Embedded Software Security through Key-Based Control Flow Obfuscation,” in *International Conference on Security Aspects in Information Technology*, vol 7011, pp. 30-44, 2011.

[28] Naoki Fujieda, Tasuku Tanaka, Shuichi Ichikawa, “Design and implementation of instruction indirection for embedded software obfuscation,” *Microprocessors and Microsystems*, vol. 45, pp. 115-128, 2016.