2019 18th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/13th IEEE International Conference On Big Data Science And Engineering

A Probability Prediction Based Mutable Control-Flow Attestation Scheme on Embedded Platforms

Jianxing Hu*†, Dongdong Huo*†§, Meilin Wang‡, Yazhe Wang*†, Yan Zhang*† and Yu Li*†
*Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China
†School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China
‡China Information Technology Security Evaluation Center
Email: hujianxing@iie.ac.cn, huodongdong@iie.ac.cn(§Corresponding Author), wang_merlin@itsec.gov.cn, {wangyazhe, zhangyan, liyu}@iie.ac.cn

***Abstract*—Control-flow attacks cause powerful threats to the software integrity. Remote attestation for control flow is a crucial security service for ensuring the software integrity on embedded platforms. The fine-grained remote control-flow attestation with execution-profiling Control-Flow Graph (CFG) is applied to defend against control-flow attacks. It is a safe scheme but it may influence the runtime efficiency. In fact, we find out only the vulnerable parts of a program need being attested at costly fine-grained level to ensure the security, and the remaining normal parts just need a lightweight coarse-grained check to reduce the overhead.**

We propose Mutable Granularity Control-Flow Attestation (MGC-FA) scheme, which bases on a probabilistic model, to distinguish between the vulnerable and normal parts in the program and combine fine-grained and coarse-grained control-flow attestation schemes. MGC-FA employs the execution-profiling CFG to apply the remote control-flow attestation scheme on embedded devices. MGC-FA is implemented on Raspberry Pi with ARM TrustZone and the experimental results show its effect on balancing the relationship between runtime efficiency and control-flow security.

***Index Terms*—Mutable Granularity Scheme, Control-Flow Attestation, Embedded Platform Security**

I. Introduction

Nowadays, a large number of embedded devices are confronted with the threat of software integrity destruction, especially the Internet-of-Things (IoT) devices. A recent study [1] shows that the embedded software suffers from a lot of vulnerabilities. Many exploitations of vulnerabilities against the software integrity on embedded devices are caused by the control-flow attack. It will influence the runtime behavior of the program. Control-Flow Integrity (CFI) [2] was proposed to defend against this threat. CFI aims to restrict the control flow of a program in a pre-defined Control-Flow Graph (CFG). For embedded devices, CFI is often deployed in the form of remote attestation [3].

Remote attestation is an effective technique for attesting the software integrity on embedded devices [4]. It consists of a resourceful trusted party (*verifier*) and a resource-constrained remote device (*prover*). The verifier gets a timely and trusted report from the prover to obtain its state. Remote attestation only keeps the necessary attestation generator on the prover and moves the costly part of the attestation architecture to the verifier. Hence, remote attestation is well-suited for embedded devices on ensuring the security and reducing the overhead.

In spite of the effectiveness of remote attestation, fine-grained control-flow attestation scheme with execution-profiling CFG on the prover still causes a large negative impact on the runtime efficiency as well as providing strong security guarantee towards the software integrity. Embedded devices are generally time critical [5], and thus this overhead cannot be ignored. In fact, we find only the vulnerable parts of a program are necessary to be attested at fine-grained level to ensure the security, and the remaining normal parts just need a coarse-grained check to reduce the overhead.

Furthermore, we can combine these two attestation schemes on a program if we can distinguish the vulnerable and normal parts in the program. For C/C++ embedded software, we use the *function* in them as a basic unit to do this combination. More specifically, if the control flow inside a function is vulnerable, such as stack overflow, the function is thought to be *vulnerable* and it will be checked at costly **fine-grained level**. This level checks all the control-flow events inside the function such as branches, calling and returning events of this function. Otherwise the function is thought to be *normal*, it will be checked at lightweight **coarse-grained level**. In our scheme, the coarse-grained level only checks the calling and returning events of this function without caring about other branches inside the function.

In this article, we propose Mutable Granularity Control-Flow Attestation (MGC-FA) scheme which uses a probabilistic model to combine fine-grained and coarse-grained control-flow attestation schemes on embedded devices. This mutable granularity refers to the changeable probability threshold, which decides the check level of the function. To be more precise, we design a machine learning model to predict the vulnerable probability p_f of each function in a program and assign a probability threshold p. Machine learning model is applied due to its elasticity and improvability. For a function with the vulnerable probability $p_f < p$, it is thought to be

2324-9013/19/$31.00 ©2019 IEEE
DOI 10.1109/TrustCom/BigDataSE.2019.00077

normal and MGC-FA performs the coarse-grained check on it. For a function with the probability $p_f \geq p$, it is predicted to be vulnerable and then it will be checked at fine-grained level. A lower threshold means more functions will be checked at fine-grained level. Two extreme thresholds are 0 and 1, which make all the functions in the program be checked at fine-grained level and coarse-grained level respectively. MGC-FA aims to combine these two schemes to reach a relative balance between efficiency and security. In MGC-FA, we can adjust the threshold p to decide the *relative* degree. To ensure the reliability of attestation, the prover should be equipped with a trusted anchor for measuring the program's runtime behavior. For ARM-based devices, such as Raspberry Pi in our experiment, TrustZone is a lightweight trusted anchor.

The static CFG is able to mitigate the *control-data attack* [6], [7], which alters the program's control data (e.g., return addresses and function pointers) to destroy the control-flow integrity, but it cannot detect the *non-control-data attack* [8]. Non-control-data attacks will corrupt the program data without introducing any invalid path in static CFG. In our scheme, CFG is defined by execution profiling, which is feasible on the embedded device due to its dedicated and simple software. We can cover the CFGs of a program with its possible inputs, then the non-control-data attack can be detected under a certain input. In this article, we use a hash value computed by *blake2* [9] to represent an execution-profiling CFG.

The main contributions of this paper are:

- We propose a scheme called MGC-FA which uses the execution-profiling CFG to detect the control-data attack and the non-control-data attack.
- MGC-FA provides a mutable granularity of control-flow attestation to reach a relative balance between runtime efficiency and security guarantee.
- We design a machine learning model to predict the vulnerable probability of each function in a program and combine both fine-grained and coarse-grained control-flow attestation schemes on embedded software.

II. Related Work

Prior works on static attestation came in three flavors: software-based, TPM-based and hybrid. TPM-based attestation [10], [11] is feasible on PCs but it is too heavy for low-end embedded devices. Software-based attestation [12], [13] fulfills the attestation with software under some strong assumptions. Hybrid scheme [14], [15] is an eclectic method which aims to minimize hardware requirements on the prover to provide strong remote attestation guarantees. In this article, our scheme captures the runtime behavior to achieve the dynamic attestation.

There are also some works on protecting the integrity of embedded software. TMDFI [16] is a hardware Data-Flow Integrity (DFI) implementation to enable fine-grained DFI checks on embedded devices. HARDWARE-ASSISTED CFI [17] presents the design of novel security hardware mechanisms for embedded devices to enable fine-grained CFI checks against runtime attacks. IB-MAC [18] is designed to protect low-cost embedded systems against malicious manipulation of their control flow as well as preventing the stack overflows.

```
void vulnerable_func(char *user, ...){
    int authenticated = 0;
    ……
    while(!authenticated){
      type = packet_read();
      if(auth_password(user, password))
          authenticated = 1;
      if(authenticated) break;
    }
    do_authenticated();
}
```

```
void normal_func(int &a, int &b){
    if(a == b) return;
    int tmp = a;
    a = b;
    b = tmp;
    return;
}
```

Fig. 1. Fine-grained check and coarse-grained check on functions

The most related work to ours is C-FLAT [4]. It introduces a fine-grained attestation scheme, which checks every branch in the program, but this fine-grained check may influence the runtime efficiency of embedded devices. Trusted virtual containers [19] allow the control-flow attestation at a coarse-grained module granularity. For C/C++ written embedded software, it is more precise to perform the coarse-grained check at function level. In our scheme, we try to combine fine-grained and coarse-grained control-flow attestation schemes to reach a relative balance between security guarantee and runtime efficiency.

III. Problem Statement and Assumptions

A. Problem Statement

In this section, we present that not all the parts of one program are necessary to be attested at fine-grained level.

We assume that the two functions in Fig. 1: *vulnerable_func* and *normal_func* are in the program *A*. In order to ensure the security, the program *A* is checked at fine-grained level. The *vulnerable_func* function contains a vulnerability: when the variable *type* reads data from the *packet_read* function, a large packet of data may trigger the overflow and pollute the variable *authenticated* to 1. It can help accessing the *do_authenticated* function without passing the check of *auth_password* function. This is the non-control-data attack. The fine-grained scheme on the *vulnerable_func* function checks each control-flow event inside the function, and thus this attack can be detected under a certain input.

We can tell that the *normal_func* function is not vulnerable. Hence, the fine-grained check on it is unnecessary and costly, in which the calling and returning events as well as the branch *if* are checked. This function only needs a lightweight coarse-grained check to reduce the overhead. Then the branch *if* will not be checked. The coarse-grained check is useful in some cases and we will discuss this in section IV-C.

Through above analysis, we know that not all the parts of one program are necessary to be attested at fine-grained level. If we can decide the vulnerable functions and normal functions in a program, both fine-grained and coarse-grained attestation schemes can be combined on the program to reach a relative balance between security guarantee and runtime efficiency. Under a proper threshold, MGC-FA scheme can detect both control-data attacks and non-control-data attacks due to the use of execution-profiling CFG.

Algorithm 1 Procedure of MGC-FA

Require: Software A; Probability threshold $p \in [0, 1]$; Key k stored in TrustZone; Possible inputs I_p for A; Specific input I_s for A; Challenge c; Probabilistic model *pm*

❶:
for each function *func* in software A **do**
$p_f \leftarrow$Predicted vulnerable probability of *func* with *pm*
end for
❷:
for each function *func* in software A **do**
if *func* probability $p_f \geq p$ **then**
func is instrumented at fine-grained level
else if *func* is detected in the re-evaluation phase **then**
func is instrumented at fine-grained level
else
func is instrumented at coarse-grained level
end if
end for
$A_p \leftarrow$Processed software A
❸: $h_s \leftarrow$Hash values of dynamic CFGs of A_p under I_p
❹: Store h_s in the database
❺: The verifier sends the challenge c to the prover
❻: When receiving c, the prover runs A_p under I_s
❼: $h \leftarrow$Hash value of dynamic CFG of A_p under I_s
❽: The prover generates a signature of the attestation report $r \leftarrow sig_k(h, c)$
❾: The prover transfers the report r to the verifier
❿: The verifier checks the validity of r in the database

B. Assumptions and Adversary Model

There are two assumptions about our scheme. Firstly, we assume the embedded system, which refers to *Raspbian* here, is thought to be trusted since the prover part of MGC-FA is built on top of the system. We only care about the software upon the system.

Secondly, there is still a possible way to circumvent the protection of dynamic CFG if an existing node in it can be overwritten. Therefore, we suppose the DEP [20] scheme, which makes the executable region read-only, is deployed in the embedded system. This scheme is a common built-in protection on the embedded platform nowadays.

In our article, we consider the adversary as an attacker who can destroy the software integrity on embedded devices by utilizing the software vulnerabilities, rather than a controller who can read/write at any memory address.

IV. Design of MGC-FA

In this section, we will introduce the MGC-FA scheme. Fig. 2 is an overview of MGC-FA, which consists of the verifier and the prover. In the verifier part, MGC-FA preprocesses software *A* to combine fine-grained and coarse-grained checks on it. The prover executes the processed program and generates an attestation report. MGC-FA aims to reach a relative balance between security and efficiency on attesting the software integrity on embedded devices.

TABLE I
The features at function level

Feature	Description
CallIn	Number of functions that call the function
CallOut	Number of functions that the function would call
CountInput	Number of inputs a function uses
CountLineCode	Number of lines containing source code
CountLineCodeExe	Number of lines containing executable code
CountOutput	Number of outputs set in a function
CountPath	Number of unique paths through a function body
Cyclomatic complexity	McCabe Cyclomatic complexity
MaxNesting	Maximum nesting level of control constructs
RatioCommentToCode	Ratio of number of comment lines to number of code lines

A. Overview of MGC-FA

Algorithm 1 depicts the procedure of MGC-FA in Fig. 2. The challenge c in this algorithm contains the identifier of the target software *A* and a nonce to ensure the freshness of the attestation report. We will explain the re-evaluation phase in section V-A.

B. Design of the Preprocessing Module

In this section we discuss about the design of the verifier part in MGC-FA. This part mainly preprocesses the target program and validates the final report.

In the preprocessing phase, the first step is to predict the vulnerable probabilities of functions in the program. In MGC-FA, we use data and features to train a machine learning model to handle this task. The data here refer to vulnerable and normal functions. Vulnerable functions can be found in many vulnerability databases such as *CVE* and *NVD*. As for the normal function, it may not exist at all since a normal function may have unknown vulnerabilities. In our scheme, if two different versions of one software step across a long period of time, the unchanged functions between them can be thought as normal functions in a high reliability. Considering two versions of software *A*, 2.4.2 (released in 2012) and 2.4.12 (released in 2015), the time gap is around 3 years, then the unchanged functions between them are thought to be normal. We also use some static analysis tools such as *cppcheck* [21] to filter the selected normal functions to ensure they are normal enough. Then features at function level are required. Some existing works [22], [23] proposed machine learning models to predict the vulnerability at method-level with various features. By using some feature selection methods [24], we choose ten features as TABLE I shows. With data and features, we train the machine learning model to predict the vulnerable probabilities of functions.

After the prediction phase, each function in the target program has a probability and they can be classified into vulnerable and normal functions with a threshold p. In previous section I, we claim that MGC-FA can also detect non-control-data attacks, but we do not put it into consideration in the prediction phase since this kind of attack has little to do with the features defined in TABLE I. Considering this and other

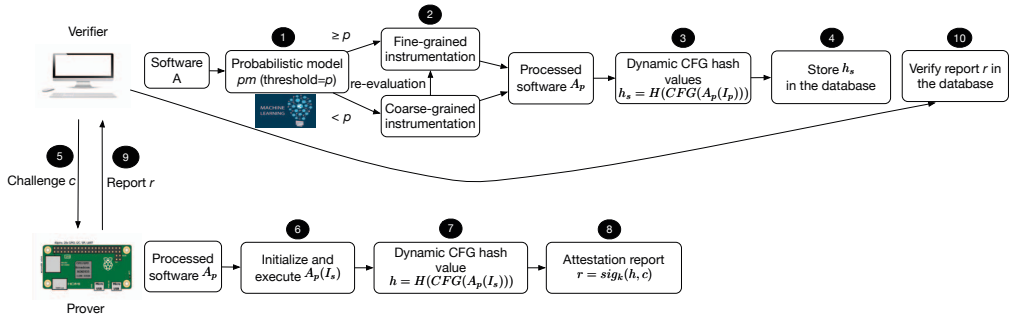

Fig. 2. Overview of MGC-FA

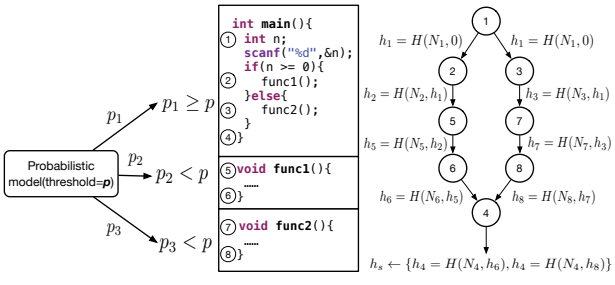

Fig. 3. The prediction and hash computation procedure

issues, we add a re-evaluation module after the prediction, which will be discussed in section V-A.

In the instrumentation phase, the target program is rewritten to perform the fine-grained check on vulnerable functions and the coarse-grained check on normal ones. After that, the processed program runs and generates various legal hash values of dynamic CFGs under possible inputs.

Fig. 3 is an example of the prediction and the hash computation procedure. The middle part is the source code with three functions: *main*, *func1* and *func2*. With a probabilistic model and a threshold p, we assume the predicted vulnerable probabilities of *main*, *func1* and *func2* are p_1, p_2 and p_3. They satisfy that $p_1 \geq p$ and $p_2, p_3 < p$. According to the scheme, the *main* function will be checked at fine-grained level and *func1*, *func2* will be checked at coarse-grained level. Four check points in the *main* function are numbered as follows: ① shows the entrance check point before branches ② and ③, ④ is the exit check point. ⑤ and ⑥ are the entrance and exit check points of *func1* in coarse-grained check while ⑦ and ⑧ belong to *func2*. This simple program has two legal hash values h_s of dynamic CFGs, which can be generated under the inputs $n \geq 0$ and $n < 0$. The hash computation scheme in the right part of Fig. 3 will be mentioned in section V-B.

C. Design of the Attestation Module

The runtime attestation part of MGC-FA shows in Fig. 4 and it is realized on the prover. The ARM architecture can be divided into the normal world and the secure world. The normal world has an *Intercepter* while the secure world contains a *Hash Management* and a *Response Generator*. The normal world can communicate with the secure world through the intercepter.

The processed target program starts to execute when receiving the challenge c from the verifier. When the execution encounters the instrumented branch, the control flow of this program will be redirected to the intercepter. In Fig. 4, there are six arrows with identifiers in the normal world. Instruction *bl A* calls the function *A*, which is checked at coarse-grained level. The 1_{in} labeled arrow captures this calling event and forwards it to the intercepter. The intercepter holds it on and delivers some information to the hash management. The hash management uses the information to compute the hash value of the dynamic CFG and returns the control flow. The 1_{out} labeled arrow captures the event of control flow returning from the intercepter to the program. Similarly, 2_{in} and 2_{out}, 3_{in} and 3_{out} are the entering and leaving events of function *A*.

When the target program finishes its execution, the hash management will pass the hash value h of the dynamic CFG to the response generator. Response generator also takes the key k and the challenge c as its parameters to generate the signature r of the attestation report and transfers it to the verifier.

In Fig. 5, we illustrate three typical attack scenes ①, ② and ③ and introduce two attacks ❶ and ❷. ❶ is the non-control-data attack, which tampers the non-control data such as *flag* from input 1 to -1 in ① and ② as well as n from input 3 to 5 in ③. ❷ is the control-data attack and it alters

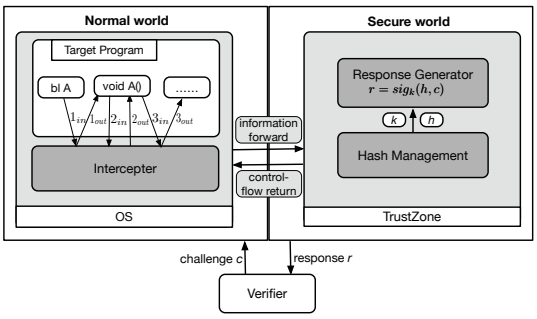

Fig. 4. The architecture of the runtime attestation part

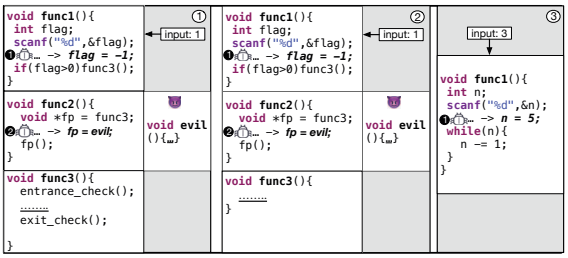

Fig. 5. Three typical attack scenes

the function pointer such as *fp* from *func3* to *evil* in ① and ②. Due to the performance of the probabilistic model and the uncertain probability threshold, it is hard to distinguish vulnerable and normal functions completely. Therefore, the vulnerable functions *func1* and *func2* may be checked at coarse-grained level and now we discuss the ability of the possible coarse-grained check on these functions. In scene ①, because of ❶ and ❷, *func3* is not directly called in *func1* and *func2*. This attack scene can be detected due to the entrance/exit check points in *func3*, which means *func3* is also instrumented by MGC-FA. Scene ② is almost the same as ① apart from *func3* is not instrumented. In scene ③, the attack tampers *n* and influences the inner loop in *func1*. The coarse-grained check cannot detect ② and ③ since it does not care about the branches inside *func1* and *func2*, while the fine-grained scheme is fit for all the three scenes. For example in scene ②, though *func3* is not checked, the calling instructions to *func3* inside *func1* and *func2* are checked at fine-grained level with dynamic CFG. Scene ① only shows the protection against calling hijacking, but it also represents the protection against return address overwrite, which is fulfilled by the exit check point. In conclusion, ① and ② represent the external function hijacking while ③ influences the inner control flow.

V. Implementation of MGC-FA

A. Implementation of the Preprocessing Module

1) Granularity Decision Phase: In this phase, we need data and features to train machine learning models. *NVD* is used as the data source of vulnerable functions. Since the vulnerability is related with the languages (C/C++) rather than the specific platform, we choose many C/C++ software packages such as *Apache httpd*, *nginx*. The vulnerability needs to be located at the function level in these programs. In this article, we use two methods to fulfill the location task: 1)Recorded vulnerability usually has a summary and some summaries contain the information about which function the vulnerability exists in. For example, the summary of CVE-2010-0010 [25]: "*Integer overflow in the ap_proxy_send_fb function in proxy/proxy_util.c...*" tells us this vulnerability is in the *ap_proxy_send_fb* function. 2)The changed functions between original and patched code are thought as vulnerable functions. With these two methods, we collect 2,100 vulnerable samples. We also gather 2,000 normal functions with the

TABLE II
Training result of various models under different thresholds

Model \ Threshold	$p = 0.45$			$p = 0.5$			$p = 0.55$		
	Pre.	Auc.	Rec.	Pre.	Auc.	Rec.	Pre.	Auc.	Rec.
RF	0.74	0.74	0.82	0.74	0.74	0.68	0.78	0.78	0.64
AdaBoost	0.61	0.63	0.92	0.76	0.76	0.68	0.63	0.61	0.23
LR	0.70	0.70	0.82	0.72	0.72	0.77	0.67	0.67	0.64

scheme mentioned in section IV-B. A tool called Understand [26] and its python interface are used to measure the features in TABLE I.

With data and features, we train three machine learning models: RandomForest (RF), AdaBoost (CART-based) and LogisticRegression (LR) to classify the vulnerable and normal functions. Generally speaking, 0.5 is the default threshold for the classification problem. In our prototype, this threshold can be adjusted according to the specific situation and we will discuss it in section VI-C. The 4,100 collected samples are divided into training data and test data in the proportion of 7:3. TABLE II shows the training result and we choose three thresholds in the experiment: the default threshold 0.5, a threshold lower than 0.5 (0.45) and a threshold higher than 0.5 (0.55). We use three metrics: Precision (*Pre.*), Area under curve (*Auc.*) and Recall (*Rec.*) to measure the model's ability on predicting the vulnerable probability. Among these metrics, recall is calculated as $Rec = \frac{TP}{TP+FN}$. TP is *True Positive* which refers to those vulnerable functions predicted to be vulnerable as well. For a vulnerable function predicted to be normal, we call this *False Negative* (FN). The higher recall is, the more vulnerable functions detected are since TP value is bigger. Thus, recall is an important metric here. TABLE II shows that lower threshold causes higher recall. If the threshold is 0, all the functions will be predicted to be vulnerable and the recall is 1. This is the safe fine-grained scheme but it may influence the runtime efficiency. Hence, it is critical to choose a proper threshold. Compared to the other two models, we choose LR as our probabilistic model due to its high recall values under three thresholds. We also compute the recall values under other thresholds and the results still show LR behaves better than the other two models.

In the re-evaluation module mentioned in section IV-B, MGC-FA reevaluates all the functions predicted to be normal in the prediction phase for two more steps. Firstly, if a variable appears in a function and its name contains keywords like *auth, key, loop*, such as the ***auth****enticated* variable in the *vulnerable_func* function in Fig. 1, the function is considered to be vulnerable. The reason for adding this step is the inner control flow of this function may be deviated due to the alteration of the key variable in it. This deviation can be detected in fine-grained check. This step requests the programmer to declare key variables containing these keywords, while it is also a chance to directly assign a function to be checked at fine-grained level if the function is thought to be important. Another step in this module is to find dangerous functions, which may introduce vulnerabilities in C/C++ programs, such

as *gets, strcpy, strcat*. If a function in the program calls any of them, it is regarded as a vulnerable function. After the re-evaluation phase, we get two function lists: Fine-Grained checked function List (FGL) and Coarse-Grained checked function List (CGL).

2) Instrumentation Phase: We rewrite the target program to hook the control-flow events for check in this phase. In order to capture the entering and leaving events of functions in CGL and FGL, we apply the *gcc* compiler to compile the source code with *-finstrument-functions* argument. This will add two instructions that call two hook functions at the entrance and exit of the function. We only need to implement these two hook functions. As for the branches inside the functions in FGL, we use Capstone [27] python interface to rewrite and hook the branches for check.

3) CFG Generation Phase: After the instrumentation phase, the target program runs and generates possible hash values of dynamic CFGs under different inputs and we store them in the database. In a real scenario, the program should run on the verifier by using software like Qemu [28] for simulating the embedded system to complete this phase.

B. Implementation of the Attestation Module

1) Intercepter: In the instrumentation phase, MGC-FA hooks the branches in the program and this would transfer the control flow of the program to the intercepter space. The intercepter calls *smc* instruction to switch from the normal mode to the secure mode and delivers some information to the hash management for computing the hash value.

2) Hash Management: This module receives information from the intercepter to compute the hash value. An example of the hash computation procedure shows in the right part of Fig. 3. We learn this hash scheme from C-FLAT [4]. In this scheme, *blake2* takes the hash value of previous path (h_x) and the node identifier in CFG (N_x) to compute the cumulative hash value. Different nodes in the CFG have different identifiers. This scheme also regards loops as subroutines and records the number of iterations, since different iterations of loops would cause an explosion in the number of hash values.

3) Response Generator: The key *k* stored in TrustZone is safe and unreachable from the normal world. Response generator uses this key *k* to generate the attestation report *r* of the hash value *h* and the challenge *c*.

VI. Evaluation

In this section, we evaluate MGC-FA scheme on its security and efficiency. MGC-FA is implemented on top of Raspbian on Raspberry Pi with TrustZone extension. Since Raspbian is an embedded linux system, we use SNU Real-Time Benchmark [29] to test MGC-FA. This benchmark has many C files, such as *adpcm-test.c* and *fft1k.c* suitable for the embedded platform. We choose some complex programs in the benchmark which have enough branches suitable for our runtime overhead experiment and introduce some vulnerabilities in them for our security experiment. MGC-FA aims to combine fine-grained and coarse-grained schemes, and thus we set three probability

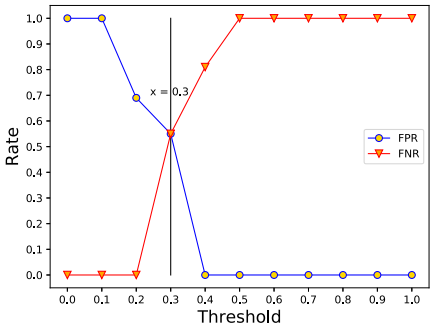

Fig. 6. The diagram of FPR and FNR

thresholds in our experiment: $p = 0$ (fine-grained check), $p = x$ (fine-grained and coarse-grained checks) and $p = 1$ (coarse-grained check).

In order to choose a proper threshold x between 0 and 1, we use our experimental data to compute *False Positive Rate (FPR)* and *False Negative Rate (FNR)*. Fig. 6 shows these two metrics under various thresholds. *FNR* is the rate of vulnerable functions predicted to be normal ($FNR = \frac{FN}{TP+FN}$) and *FPR* is the rate of normal functions predicted to be vulnerable ($FPR = \frac{FP}{TN+FP}$). *FPR* reflects the efficiency of our scheme since a higher *FPR* means more normal functions are predicted to be vulnerable and checked at costly fine-grained level. Similarly, *FNR* reflects the security of MGC-FA. In this experiment, we try to keep an equal balance between efficiency and security. Therefore, both of these two metrics should be kept at low values. Fig. 6 tells that threshold at $p = 0.3$ keeps both *FPR* and *FNR* relatively low. Hence, we choose x as 0.3 in our experiment to combine fine-grained and coarse-grained schemes.

A. The Evaluation of Runtime Overhead

The runtime overhead experiment result shows in TABLE III. In order to better illustrate the overhead change, the *Runtime overhead* metric is expressed in magnification times compared to the corresponding initial program overhead. All the initial overheads are set to 1 and they are not displayed in TABLE III.

We choose the first example *adpcm-test.c* for illustrating, whose runtime overhead is measured under four states. First we run the initial program for multiple times to obtain its average runtime (0.02s) and this overhead is set as the initial overhead 1. Then MGC-FA is applied with $p = 0$ and all the 19 functions in the program are predicted to be vulnerable (*normal/vulnerable*=**0/19**). Thus, they are checked at fine-grained level. This runtime overhead expands by about **600** times (12s) and the number of checked control-flow events is up to $\mathbf{2 \times 10^6}$. Next we raise the threshold p up to 0.3 and 7 out of 19 functions are checked at fine-grained level (*normal/vulnerable*=**12/7**), which reduces the overhead to **25** times (0.5s) with checking about $\mathbf{1.1 \times 10^5}$ events. Since the

TABLE III
RUNTIME OVERHEAD UNDER DIFFERENT THRESHOLDS

File/Software \ Metric	Predicted functions (normal/vulnerable)			Runtime overhead (times)			Checked control-flow events (number)		
	MGC-FA Prototype			MGC-FA Prototype			MGC-FA Prototype		
	$p=0$	$p=0.3$	$p=1$	$p=0$	$p=0.3$	$p=1$	$p=0$	$p=0.3$	$p=1$
adpcm-test.c	0/19	12/7	19/0	600	25	23	2×10^6	1.1×10^5	1×10^5
fft1k.c	0/6	5/1	6/0	462	333	39	3.6×10^5	2.7×10^5	3.3×10^4
fir.c	0/8	6/2	8/0	439	133	36	7800	2400	734
lms.c	0/8	6/2	8/0	476	124	25	1.2×10^5	4.4×10^4	8912
ludcmp.c	0/3	2/1	3/0	379	284	10	574	410	14
qurt.c	0/4	3/1	4/0	333	288	25	248	202	14
minver.c	0/4	1/3	4/0	208	187	3	310	300	6
fft1.c	0/6	2/4	6/0	398	320	53	884	776	136
sqrt.c	0/3	3/0	3/0	301	10	9	296	4	4

TABLE IV
SECURITY GUARANTEE UNDER DIFFERENT THRESHOLDS

File/Software	Vulnerable function	Vulnerable probability (p_f)	Attack scene	Detectable		
				p=0	p=0.3	p=1
adpcm-test.c	encode	0.20	*scene#1*	Yes(f)	Yes(c)	Yes(c)
fft1k.c	fft_c	0.26	*scene#3*	Yes(f)	No(c)	No(c)
fir.c	sin	0.31	*scene#1*	Yes(f)	Yes(f)	Yes(c)
lms.c	lms	0.16	*scene#2*	Yes(f)	No(c)	No(c)
ludcmp.c	ludcmp	0.39	*scene#3*	Yes(f)	Yes(f)	No(c)
qurt.c	qurt	0.32	*scene#1*	Yes(f)	Yes(f)	Yes(c)
minver.c	fabs	0.23	*scene#1*	Yes(f)	Yes(c)	Yes(c)
fft1.c	fft1	0.40	*scene#3*	Yes(f)	Yes(f)	No(c)
sqrt.c	sqrt	0.28	*scene#3*	Yes(f)	Yes[1](f)	Yes[1](f)
OSP	readSerial	0.27	*scene#1*	Yes(f)	Yes(c)	Yes(c)
OSP	bolus	0.40	*scene#3*	Yes(f)	Yes(f)	No(c)

number of checked events at $p=1$ ($\mathbf{1\times10^5}$) almost remains the same as $p=0.3$ ($\mathbf{1.1\times10^5}$), the runtime overhead does not reduce much at $p=1$ (**23**).

We can see from the *Runtime overhead* metric in TABLE III that the runtime efficiency at threshold $p=0.3$ behaves better than the fine-grained scheme ($p=0$) and worse than the coarse-grained one ($p=1$).

B. The Evaluation of Security Guarantee

In this section, we introduce some vulnerabilities into the functions to validate the security guarantee of MGC-FA. TABLE IV shows the details about the predicted *vulnerable probability* (p_f) of the *vulnerable function* in *File/Software* and whether various *attack scenes* can be detected under different thresholds in MGC-FA. The *(f)* mark in *Detectable* column means the corresponding *Vulnerable function* is checked at fine-grained level under this threshold p ($p_f \geq p$) while *(c)* is the coarse-grained check ($p_f < p$). We add another example Open Syringe Pump (OSP) from C-FLAT [4].

In the *Attack scene* column, there are three kinds of attack scenes: *scene#1*, *scene#2*, *scene#3* corresponding to scenes ①, ② and ③ respectively in Fig. 5. TABLE IV indicates *scene#1* is detectable in both fine-grained and coarse-grained schemes, while *scene#2* and *scene#3* can only be detected in fine-grained check. We can see that the *scene#1* case exists in a real scenario. These results are consistent with the abilities discussed in section IV-C. At threshold $p=0.3$, the number of attacks MGC-FA can detect (9 *Yes*) is between the $p=1$ (6 *Yes*) and $p=0$ (11 *Yes*). Taking into account the runtime overhead experiment, we can see MGC-FA combines fine-grained and coarse-grained schemes and reaches a relative balance between efficiency and security.

We choose OSP, a real software package on embedded devices for explaining. This software controls a stepper motor which is connected to a syringe pump in order to deliver precise quantities of liquid from the syringe. It has two main CFGs and we can get limited hash values due to our hash scheme. OSP contains two main control-flow vulnerabilities: *scene#1* and *scene#3*. The *scene#1* vulnerability in OSP is caused by the global fixed-length array *serialStr* in *readSerial* function.

Scene#3 is the non-control-data attack and it tampers a variable that controls a loop in *bolus* function. This function is checked at fine-grained level since its probability $p_f=0.4$ is higher than the threshold $p=0.3$. Due to the hash scheme, this attack can be detected in fine-grained check because of the different iterations from the original one in the database.

C. Further Research on Proper Threshold

In our experiment, we use the threshold $p=0.3$ to combine fine-grained and coarse-grained schemes. But how to set a proper threshold in an actual scenario without the knowledge of *FPR* and *FNR* is a critical issue. In this part, we will do the following research on this problem.

We collect another 100 software packages and each of them has many functions, including known vulnerable functions and known normal functions. We compute FPR and FNR for each program under various thresholds to get the intersection threshold x of these two metrics. This intersection threshold aims to get an equal balance between efficiency and security like the threshold $p=0.3$ in Fig. 6. Altogether we get 100 intersection thresholds. The mean value $\bar{X}$ and the standard deviation S of these 100 intersection thresholds are 0.32 and 0.13 respectively. Then we use Equation (1) to obtain a 95% confidence interval (0.29, 0.35) with $Z_{(1-\alpha)}=1.96$ [30]. The n in Equation (1) is the sample size 100. Hence, the intersection threshold of a new program will lie in the interval (0.29, 0.35) in a high probability. This confidence interval is related to the performance of our current probabilistic model.

$$(\bar{X}-Z_{(1-\alpha)}\times\frac{S}{\sqrt{n}},\bar{X}+Z_{(1-\alpha)}\times\frac{S}{\sqrt{n}}) \quad (1)$$

In our experiment, $p=0.3$ is a tradeoff threshold but in some occasions we focus on efficiency or security. If a program needs more security guarantee, the threshold should be decreased. In Fig. 6, we can see the *FPR* increases and *FNR* decreases when the threshold becomes lower, since more vulnerable functions will be checked at fine-grained level. Similarly, in order to improve the efficiency, the threshold should be increased. For a new program, we can firstly try the threshold in the interval (0.29, 0.35) to keep an equal balance between efficiency and security, and then adjust it according to the specific need (more security or higher efficiency).

[1]This non-control-data attack is caused by a variable *key_lp*. The *sqrt* function is regarded to be vulnerable in the re-evaluation phase.

VII. Conclusion and Discussion

In this article, we propose MGC-FA to solve the costly problem of the fine-grained control-flow attestation scheme and employ a coarse-grained scheme to reduce the overhead. MGC-FA combines both fine-grained and coarse-grained control-flow attestation schemes on embedded devices by using a probabilistic model. This prototype can detect control-flow deviations under a proper threshold, including control-data attacks and non-control-data attacks. MGC-FA is proved to reach a relative balance between security guarantee and runtime efficiency, but it also has some drawbacks.

First, MGC-FA needs the source code to predict the vulnerable probability of each function depending on source code features. For those embedded software packages which are not open-source, we cannot apply our prototype. Second, MGC-FA currently aims at the program written in C/C++ languages but for other languages that do not use the function as their basic unit such as Java, MGC-FA is not applicable.

The future work will focus on two aspects. First, we will try to improve the performance of our probabilistic model. Our scheme will perform better as our model can predict more accurate vulnerable probabilities. This means our probabilistic model can predict a higher probability for the vulnerable function than the normal function. Second, we will propose an algorithm to adjust the probability threshold automatically in case of the specific need. In this algorithm, the verifier is required to assign an expected runtime upper bound t for a program, and then the algorithm will change the threshold to meet the efficiency requirement and keep the security as much as possible. For example, suppose that the program's runtime t_r satisfies $t_r > t$ when MGC-FA performs the check on the program under an initial threshold 0.3, then this algorithm will do a binary search in the threshold interval (0.3, 1] to increase the threshold to 0.65 (reduce the overhead) and vice versa. This procedure will not stop until the threshold interval is small enough. This algorithm will work out a lowest threshold that meets the overhead need ($t_r \leq t$) and keeps the security as much as possible. As for the security guarantee degree, it depends much on the performance of the probabilistic model.

ACKNOWLEDGMENT

This work is supported by the National Key R&D Program of China (2017YFB0801902).

References

[1] A. Costin, J. Zaddach, A. Francillon, and D. Balzarotti, "A large-scale analysis of the security of embedded firmwares," in *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, 2014, pp. 95–110.
[2] M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," 2005.
[3] M. Santra, S. K. Peddoju, A. Bhattacharjee, and A. Khan, "Design and analysis of a modified remote attestation protocol," in *2017 IEEE Trustcom/BigDataSE/ICESS*. IEEE, 2017, pp. 578–585.
[4] T. Abera, N. Asokan, L. Davi, J.-E. Ekberg, T. Nyman, A. Paverd, A.-R. Sadeghi, and G. Tsudik, "C-flat: control-flow attestation for embedded systems software," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 743–754.
[5] S. Ravi, A. Raghunathan, P. Kocher, and S. Hattangady, "Security in embedded systems: Design challenges," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 3, no. 3, pp. 461–491, 2004.
[6] M. Tran, M. Etheridge, T. Bletsch, X. Jiang, V. Freeh, and P. Ning, "On the expressiveness of return-into-libc attacks," in *International Workshop on Recent Advances in Intrusion Detection*. Springer, 2011, pp. 121–141.
[7] H. Shacham *et al.*, "The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86)." in *ACM conference on Computer and communications security*. New York,, 2007, pp. 552–561.
[8] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer, "Non-control-data attacks are realistic threats." in *USENIX Security Symposium*, vol. 14, 2005, p. 12.
[9] J.-P. Aumasson, S. Neves, Z. Wilcox-OHearn, and C. Winnerlein, "Blake2: simpler, smaller, fast as md5," in *International Conference on Applied Cryptography and Network Security*. Springer, 2013, pp. 119–135.
[10] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig, "Trustvisor: Efficient tcb reduction and attestation," in *2010 IEEE Symposium on Security and Privacy*. IEEE, 2010, pp. 143–158.
[11] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki, "Flicker: An execution infrastructure for tcb minimization," in *ACM SIGOPS Operating Systems Review*, vol. 42, no. 4. ACM, 2008, pp. 315–328.
[12] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. Van Doorn, and P. Khosla, "Pioneer: verifying code integrity and enforcing untampered code execution on legacy systems," in *ACM SIGOPS Operating Systems Review*, vol. 39, no. 5. ACM, 2005, pp. 1–16.
[13] A. Seshadri, A. Perrig, L. Van Doorn, and P. Khosla, "Swatt: Software-based attestation for embedded devices," in *IEEE Symposium on Security and Privacy, 2004. Proceedings. 2004*. IEEE, 2004, pp. 272–282.
[14] K. Eldefrawy, G. Tsudik, A. Francillon, and D. Perito, "Smart: Secure and minimal architecture for (establishing dynamic) root of trust." in *NDSS*, vol. 12, 2012, pp. 1–15.
[15] P. Koeberl, S. Schulz, A.-R. Sadeghi, and V. Varadharajan, "Trustlite: A security architecture for tiny embedded devices," in *Proceedings of the Ninth European Conference on Computer Systems*. ACM, 2014, p. 10.
[16] T. Liu, G. Shi, L. Chen, F. Zhang, Y. Yang, and J. Zhang, "Tmdfi: Tagged memory assisted for fine-grained data-flow integrity towards embedded systems against software exploitation," in *2018 17th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/12th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*. IEEE, 2018, pp. 545–550.
[17] L. Davi, P. Koeberl, and A.-R. Sadeghi, "Hardware-assisted fine-grained control-flow integrity: Towards efficient protection of embedded systems against software exploitation," in *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE, 2014, pp. 1–6.
[18] A. Francillon, D. Perito, and C. Castelluccia, "Defending embedded systems against control flow attacks," in *Proceedings of the first ACM workshop on Secure execution of untrusted code*. ACM, 2009, pp. 19–26.
[19] K. A. Bailey and S. W. Smith, "Trusted virtual containers on demand," in *Proceedings of the fifth ACM workshop on Scalable trusted computing*. ACM, 2010, pp. 63–72.
[20] (2019) DEP Protection. [Online]. Available: http://t.cn/EVkSDF0
[21] D. Marjamäki, "Cppcheck: a tool for static c/c++ code analysis," 2013.
[22] E. Giger, M. D'Ambros, M. Pinzger, and H. C. Gall, "Method-level bug prediction," in *Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement*. ACM, 2012, pp. 171–180.
[23] H. Hata, O. Mizuno, and T. Kikuno, "Bug prediction based on fine-grained module histories," in *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 2012, pp. 200–210.
[24] G. Chandrashekar and F. Sahin, "A survey on feature selection methods," *Computers & Electrical Engineering*, vol. 40, no. 1, pp. 16–28, 2014.
[25] (2019) CVE-2010-0010. [Online]. Available: http://t.cn/EKIQ5y5
[26] (2019) Scitools Homepage. [Online]. Available: https://scitools.com/
[27] (2019) Capstone Homepage. [Online]. Available: http://t.cn/8kwsx88
[28] F. Bellard, "Qemu, a fast and portable dynamic translator." in *USENIX Annual Technical Conference, FREENIX Track*, vol. 41, 2005, p. 46.
[29] (2019) SNU Homepage. [Online]. Available: http://t.cn/EIhojqQ
[30] (2019) t-Table. [Online]. Available: http://t.cn/EINGLbM