Hindawi
Security and Communication Networks
Volume 2021, Article ID 2914192, 16 pages
https://doi.org/10.1155/2021/2914192

Research Article

NSGA-II-Based Granularity-Adaptive Control-Flow Attestation

Jing Zhan,[1,2] **Yongzhen Li**,[1,2] **Yifan Liu**,[1,2] **Hongchao Li**,[1,2] **Shuai Zhang**,[1,2] **and Li Lin**[1,2]

[1]*School of Computer Science, Faculty of Information Technology, Beijing University of Technology, Beijing 100124, China*
[2]*Beijing Key Laboratory of Trusted Computing, Beijing 100124, China*

Correspondence should be addressed to Jing Zhan; zhanjing@bjut.edu.cn, Yongzhen Li; liyongzhen97@163.com, and Li Lin; linli_2009@bjut.edu.cn

Received 17 September 2021; Accepted 27 October 2021; Published 18 November 2021

Academic Editor: Chunhua Su

Since the widespread adoption of edge computing and IoT technology, Control-Flow Hijacking (CFH) attacks targeting programs in resource-constrained embedded devices have become prevalent. While the Coarse-Grained Control-Flow integrity Attestation (CGCFA) lacks accuracy for the CFH attacks detection, the Fine-Grained Control-Flow integrity Attestation (FGCFA) detect the attacks more accurately but with high overheads, which can be a big burden (e.g., to industrial control system with strict performance requirements). In this paper, we propose a NSGA-II (Nondominated Sorting Genetic Algorithm-II) based Granularity-Adaptive Control-Flow Attestation (GACFA) for the programs in embedded devices. Specifically, we propose a Granularity-Adaptive Control-Flow representation model to reduce the complexity of programs' control-flow graph and propose NSGA-II-based granularity-adaptive strategy generation algorithm to balance the security and performance requirements. Besides, runtime protection for the GACFA at the program end with SGX is proposed to protect the integrity and confidentiality of control-flow measurement data. The experiments show that our work can find out the best-so-far control-flow granularity with stability and provide secure program attestation for the verifier. In addition, the security/performance benefit of adopting our proposal over CGCFA is 13.7, 25.1, and 43.0 times that of adopting FGCFA over ours in different threat scenarios.

1. Introduction

With the rapid development of edge computing and IoT technology, more and more embedded devices are connected together and reach people's daily works and lives, which brings us both convenience and security concerns. For example, the connection between the industrial control system which contains lots of resource-constrained embedded devices and the corporate intranet or the Internet has increased the attack surface, leading to more remote attacks [1]. In recent years, Control-Flow Hijacking (CFH) attacks, which can directly tamper with the behaviour of programs running in the memory, make it a challenge to ensure the security of programs of embedded devices.

Remote attestation is a method of verifying the integrity of the software on a remote device. At the end of the attestation, the verifier obtains a report signed by a security chip (e.g., Trusted Platform Module, TPM) inside the device from the prover on whether the hardware and software codes' hashes to be executed meet the verifier's expectations. Later, the research on memory verification and attestation of the hosts [2] and embedded devices [3] is proposed too. However, these memory verification and attestation works can only verify the integrity of the whole memory chuck but cannot find out whether the control flow of specific program running in the memory is hijacked.

In recent years, practical control-flow integrity research has been proposed to ensure runtime program integrity by comparing whether the destination address of the jump instruction is in the branch target sets recorded by pre-analysis [4]. However, the method of comparing address and target sets can only find out the attacks tampering with the control data (the called function pointer, the address pointed to by the jump instruction inside the function, etc.) but cannot detect the hijacking attack on the program control flow by tampering with the noncontrol data (such as branch

and loop variable value) inside the function [5]. Later, the fine-grained control-flow integrity measurement and attestation [6, 7] are proposed to verify runtime program control flow more accurately, since it obtains more context information in the granularity of basic block, but with increased overheads, which is not efficient for providing real-time and reliable services in mission critical systems, for example, industrial control system.

In order to balance the security and efficiency of CFH attacks detection in a resource-constrained embedded environment, this paper proposes NSGA-II (Nondominated Sorting Genetic Algorithm-II) based Granularity-Adaptive Control-Flow Attestation (GACFA), which take functions and basic blocks as different control-flow monitoring granularities. Through genetic algorithm NSGA-II, basic block-level fine-grained control-flow monitoring is performed on core functions that have a greater impact on program security, and function-level control-flow monitoring is performed on a noncore function. In this way, the verifier can verify in runtime whether the program has suffered CFH attacks with balanced security and overhead costs.

The contributions of this paper are as follows:

(1) A Granularity-Adaptive Control-Flow representation model for CFH detection is proposed, in which the granularity of function and basic block and virtual nodes are introduced to reduce the complexity of programs' control-flow graph and to mitigate verification path explosion problem.

(2) A NSGA-II-based granularity-adaptive strategy generation method on multiobjective optimization algorithm is proposed, which combines low-overhead and high control-flow security as the optimization goal. With this method, the optimal granularity division strategy is generated, achieving the balance between CFH detection accuracy (e.g., the security goal) and detection overheads (e.g., the performance goal).

(3) The Granularity-Adaptive Control-Flow Attestation protected by the SGX technology is also proposed to protect the integrity and confidentiality of control-flow measurement data on the program-end environment.

(4) We also implement a proof of concept system of GACFA. The experiments show that our work can find the best-so-far control-flow granularity with stability and provide secure program attestation for the verifier. The security/performance benefit of adopting our proposal over CGCFA is 13.7, 25.1, and 43.0 times that of adopting FGCFA over ours in different threat scenarios.

2. Related Works

Research on program verification and attestation at the control-flow level is mainly in two fields, including Control-Flow Integrity (CFI) detection/prevention and Control-Flow integrity Attestation (CFA).

The CFI detection/prevention is proposed to detect/prevent the hijacking on program control flow by checking it with the original control-flow graph (CFG) or predefined control-flow policy (e.g., matching set of targets for each indirect control transfer). The GRIFFIN [8] provided online CFI enforcement over unmodified binaries by checking captured control flows with predefined control-flow policies with the aid of the hardware. CFI based on a Lightweight Encryption Architecture with Advanced Encryption Standard (LEA-AES) [9] was proposed to ensure its security through encryption. Burow et al. [10] introduced and compared a broad range of CFI mechanisms and discussed them from the aspects of precision, security, and performance. In order to improve the accuracy, lots of research was proposed to use more context information of CFI detection. Ding et al. [11] proposed a path-sensitive CFI scheme, which used path-sensitive points at runtime to calculate the legal control transfer target. The value-based constraint CFI (vCFI) [12] was proposed to improve the effectiveness of CFI by monitoring and protecting control and noncontrol data that may be manipulated for Control-Flow Hijacking. The Origin-Sensitive CFI (OS-CFI) [13] was proposed to improve the security of CFI with new contextual information, such as the last branches taken. Khandaker et al. [14] also proposed Control-Flow Integrity with Backtracking (CFI-LB) with adaptive call site sensitivity, which improves program security. Jang et al. [15] proposed an Index-based Bit Vector Control-Flow Integrity (IBV-CFI) scheme to reflect the CFG with accuracy. All in all, the current CFI schemes (including control-flow capture and integrity verification) with more context information are costly and thus expensive for resource-constrained embedded devices.

Remote attestation allows a resource-rich verifier to obtain the program status of the prover with limited resources, which can reduce the CFI verification overhead for embedded devices. The traditional remote attestation schemes are mostly static or coarse, which can only guarantee the integrity of the binary code before the program runs or guarantee the security of the whole embedded devices' memory [3]. Later, C-FLAT (Control-Flow Attestation) [6] was proposed to the verifier with the proof of the application's control-flow path on the prover's device using ARM TrustZone hardware. Then, LO-FAT (Low-Overhead Control-Flow Attestation) [16] was introduced as a hardware version of C-FLAT with lower performance overhead. A runtime remote attestation system ATRIUM [7] was also proposed to attest both the code's binary and its execution behaviour to resist Time of Check Time of Use (TOCTOU) attacks. In addition, ATRIUM provided the method to protect codes' control flow and CFG with the help of the FPGA. Although the hardware-based CFA improves the security of the attestation mechanism, it means extra hardware support and costs.

The SGX (Software Guard Extensions) is an instruction set extension supported by Intel CPU since 2013 [17, 18]. The SGX can provide applications with a trusted execution environment for many scenarios now, such as secure cloud computing [19] and machine learning [20]. SGX is also widely adopted for code and data confidentiality protection for

endpoint systems and devices with relatively cheap performance costs compared to other customized secure hardware because it is a Commercial Off-The-Shelf (COTS) solution. For example, Wang et al. [21] proposed security-enhanced attestation between IoT terminals and devices with SGX-protected attestation codes and keys with about additional 3% time costs. SGX-Log [22] was proposed to protect the system logging program and log data's HMAC key using SGX enclave and sealing primitives with an overhead of 4.84% in log generation and 6.29% in log reading.

3. Threat Model and Assumptions

The existing CFH attacks can be categorized into two groups of attacks, where one is attacks tampering control data and the other is attacks tampering noncontrol data of program control flows. Specifically speaking, the control data includes the return address pointed to by the indirect jump instruction, function pointer, saved return address of functions, and so on. Noncontrol data can be divided into control flow-related variable data, such as branches and loop variables that affect program control flow, and pure data, such as program variables that have nothing to do with control flow. This paper mainly studies control data and control flow-related noncontrol data attacks detection and attestation.

We assume that, in a running program's memory, the code segment can be read and executed, while data and stack/heap segment can only be read and written. This protection is already widely adopted and can ensure that malicious code cannot be executed in the data and stack/heap segment. Figure 1 shows four types of control-flow attacks that the attackers may launch on a program that diverts the original path of the program to an unexpected one. The control-flow graph on the right side represents the control flow of the program running in the memory.

Attack 1 (branch variable attacks): this type of attack achieves the purpose of the attack by tampering with the variables that control the direction of the branch. The attack makes the program control flow going to a branch different from the original one. For example, as the CFG in Figure 1 shows, the attacker tampers with the condition variable in node 1, causing the control flow that should have jumped to node 2 to jump to node 6, making the control flow to an unexpected but legal path.

Attack 2 (loop variable attack): this attack tampers with the loop variable to change the number of loops. For example, the attacker may skip the loop of node 2 by changing the cyclic variable that controls the number of program loops to 0.

Attack 3 (return address attack): this attack alters the return address of functions to make the program control flow to the wrong place. For example, after function calling, node 3 should return to node 2, but if the return address is tampered with, the program returns to node 5 instead.

Attack 4 (function pointer attack): this kind of attack makes the program control flow go to the wrong direction by tampering with the function pointer. For example, after the execution of node 4, the program should call the function going to node 5, but the attacker tampered with the calling function pointer to cause the program to turn to node 7.

We also give the following three assumptions for our method.

(1) GACFA detects attacks that affect program control flow but does not detect pure data-driven attacks. Because pure data attacks are related to specific applications, they often need to be jointly considered with other methods.

(2) Attackers cannot physically tamper with the program code while the program is running. We assume it is more difficult for the attackers to tamper with the hardware device than with embedded software. In some cases, embedded devices (e.g., industrial control equipment) usually receive certain physical protection and are difficult for an attacker to tamper with physically.

(3) The verifier has enough memory space to store the control-flow information of the program to be verified and thus can help ensure the integrity and confidentiality of the verification process. It is assumed that the verifier has more resources and thus more security mechanisms (e.g., could be SGX) than the embedded device to be verified.

4. System Architecture

This paper proposes a Granularity-Adaptive Control-Flow Attestation (GACFA) method based on NSGA-II [23]. In this way, we extend the traditional static binary attestation and the CFA at the control-flow level to the dynamic runtime attestation with adaptive granularity. As shown in Figure 2, our system architecture consists of two roles: the verifier and the prover. The verifier is responsible for offline verification of the program to be verified, including a granularity division module and a control-flow measurement module. The online attestation module is deployed between the verifier and the prover. In addition, the prover is also in charge of collecting control-flow information of the program and prover-end control-flow measurements calculation.

4.1. Granularity Division Module. We regard the function as the high-level unit for program execution, and we mark every function as the core function or noncore function according to the importance of the function to the program. Then, we perform basic block-level control-flow monitoring for core functions and perform function-level control-flow monitoring for noncore functions.

Definition 1 (core function). It refers to the function that has a strong impact on the whole program. The more the function is called, the more important the function is to the program, which implies it is easier to be exploited by an attacker.

Definition 2 (noncore function). It refers to the function that has less impact on the whole program.

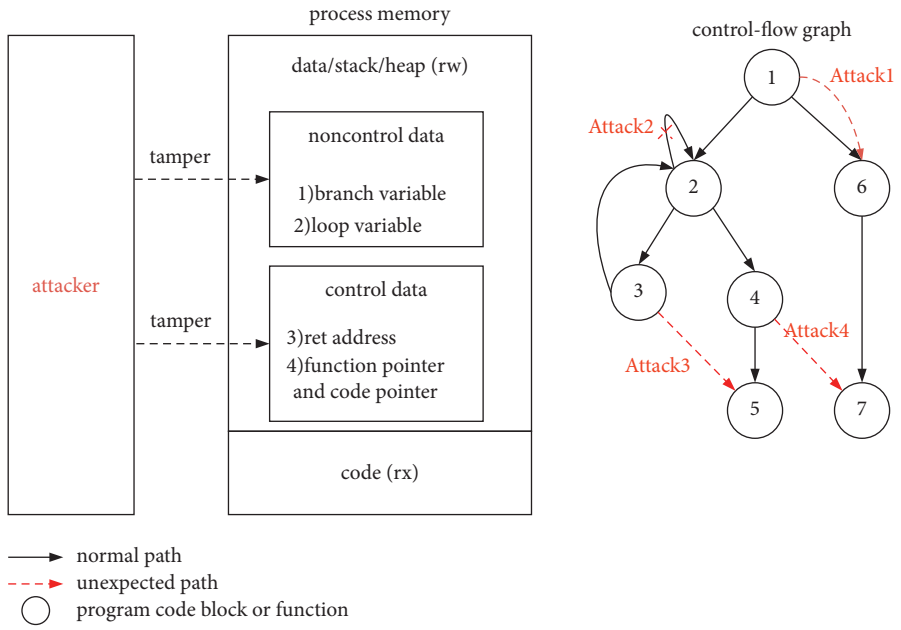


Figure 1: Threat model.

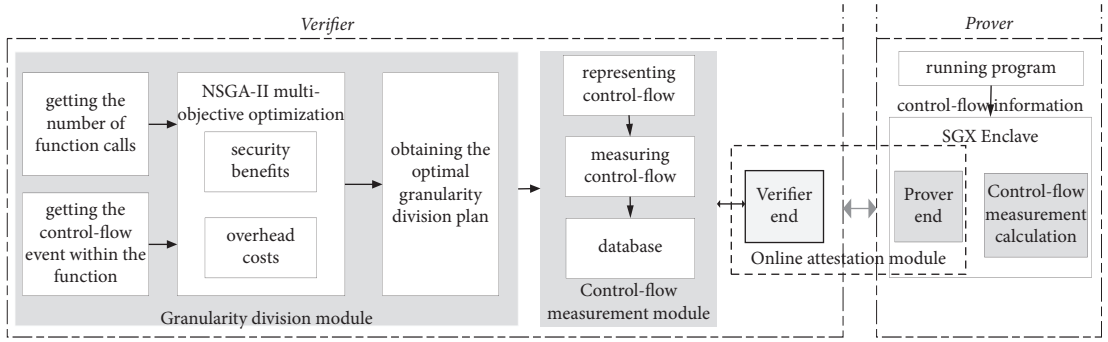


Figure 2: System architecture of GACFA.

Granularity division can be regarded as a combined optimization problem, which integrates two goals, that is, reducing overheads and improving control-flow security. The granularity division module includes three parts: input generation, target optimization, and strategy selection.

(1) Input generation: the program A to be verified is usually composed of multiple functions. Firstly, the verifier obtains the function set $F(A)$ contained in program A with static analysis. Then the verifier dynamically analyses the execution process of program A under a specific input (or no input). For function f_i in $F(A)$, count the number of times f_i is called, and the number of control-flow events contained in f_i. Here, the number of control-flow events is the number of the basic blocks in the function. At last, we take the number of f_i calls and the number of control-flow events as the input of the NSGA-II algorithm, where the number of function calls represents the degree of the security impact of the function on the entire program, and the number of control-flow events represents the overhead.

(2) Target optimization: we use NSGA-II multiobjective optimization method to evaluate the combined target, which is composed of control-flow security and overhead. After determining the control-flow granularity of the program, its security benefits are measured. And the overhead refers to the time required to authenticate the program's control flow.

(3) Strategy selection: we select the optimal core function marking scheme according to the optimization result, in order to perform fine-grained measurement on core functions and coarse-grained measurement on noncore functions later.

4.2. Control-Flow Measurement Module. The control-flow measurement module is responsible for calculating the control-flow hash value of the program to be verified, which

is divided into control-flow representation, control-flow measurement, and the measurement database.

(1) Control-flow representation: according to the division results, we perform basic block-level instrumentation on the core function and function-level instrumentation on the noncore function. Then, we obtain the control-flow information that combines the coarse and fine granularity of the program.

(2) Control-flow measurement: after control-flow representation, we can get the granularity-adaptive CFG of the program. The type of the node in the CFG can be function node, basic block node, and so on. Then, we generate the expected measurement value along the path of the CFG with accumulated hashing. The calculation formula is as follows:

$$\begin{cases} H(A_1) = H(H(0), A_1) & i = 1, \\ H(A_i) = H(H(A_{i-1}), A_i) & i > 1 \,\&\&\, \text{type}(A_i) = \text{type}(A_{i-1}). \end{cases} \tag{1}$$

Here, A_i is the node i in program's CFG. $H(0)$ is the initial hash value, which is all zero. There are three types of nodes, including function node, basic block node, and virtual node, which is specified in Section 5. For each path in the CFG, the path is divided into several parts when the node type changes. For a part in the path, the hash value of the current node and later node are taken as the inputs of the SHA-256 hash algorithm to get the accumulated hash value until all nodes are processed.

As formula (2) shows, after every parts' hash value is calculated, all parts' hash value can be accumulated again in the way similar to formula (1), and the final hash value of every path is the measurement of current program control flow. In this way, if the final hash value verification failed, we can find out which part is wrong quickly:

$$\begin{cases} H(P_1) = H(H(0), P_1) & i = 1, \\ H(P_i) = H(H(P_{i-1}), P_i) & i > 1. \end{cases} \tag{2}$$

(3) Database: it stores all the parts and paths (in the CFG of the program to be attested) accumulated hash value corresponding to program A in the measurement database.

4.3. Online Attestation Module. The online attestation module contains the verifier-end and the prover-end. As the prover obtains the program's runtime control-flow information, it generates and submits the signed control-flow report to the verifier for runtime verification. At the prover-end, Intel SGX provides a trusted execution environment for the prover to calculate the final hash value on the program's execution path and ensures the integrity of the report. The remote attestation protocol is as shown in Figure 3 and the following description.

Step 1: the verifier sends a challenge to the prover to indicate the attestation request, including the *id* of the program to be authenticated, the random number N, and the input i of the program. Among them, the random number is to ensure the freshness of the attestation result and prevent replay attacks.

Step 2: after receiving the attestation request from the verifier, the prover initializes the attestation mechanism and runs program A under input i. The control flow of the program is sent to the measurement module in the SGX enclave with the help of the runtime tracing tool (e.g., Intel Pin). The obtained control-flow information is calculated with formulas (1) and (2), and the final hash value of program's control flow and N are signed by the private key sk of the enclave for attestation report r generation.

Step 3: the prover sends signed r and the measurement h to the verifier.

Step 4: the verifier uses the public key pk to verify the signature after receiving the attestation report. If the verification succeeds, the verifier continues to check whether the control-flow measurement is consistent with the expected value under input i stored in the database. If the result is true, it means that program A has not been attacked by Control-Flow Hijacking.

4.4. SGX Protected Control-Flow Measurement Module. We use the SGX to create a trusted execution environment for control-flow measurement calculation and the prover-end attestation to protect the measurement calculation and attestation report generation.

The prover-end attestation module is already introduced in Section 4.3. And the control-flow measurement calculation module gets control-flow information from the running program and then calculates the hash value of the program execution path.

5. Granularity-Adaptive Control-Flow Representation Model

In order to perform Control-Flow Attestation, the verifier asks the execution path of the program to be measured from the prover's device. Recording and transmitting each executed instruction in the execution path is not feasible because it will lead to very long and nested paths requiring the prover to be calculated and the verifier to traverse, which is costly to attestation. To reduce the complexity of the program's control-flow graph and balance the overheads and the security of Control-Flow Attestation, this paper proposes a granularity-adaptive control flow representation model, which provides basic block-level monitoring for core functions and function-level monitoring for noncore functions. The specific definition of the model is introduced below.

Definition 3. Directed graph of the control flow of a program, denoted as G. $G = \langle V, E, A \rangle$. The model of granularity-

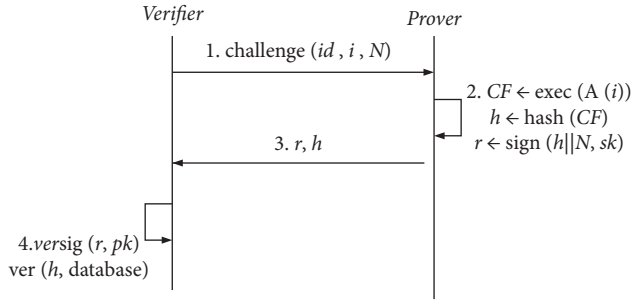


Figure 3: Online attestation protocol.

adaptive control flow is a directed graph representing program control-flow information, which is defined as a triplet $\langle V, E, A\rangle$.

Definition 4. Vertices set of G, denoted as V. $V = \{v_i | i \in N\}$. V is the set of vertices in G and each vertex represents a control-flow node, which can be a function, basic block, or virtual node.

Definition 5. Edge set of G, denoted as E. $E = \{v_i \longrightarrow v_{i+1} | i \in N\}$. E is the set of edges in G, which is used to represent the call, jump, and return relationship between node v_i and v_{i+1}.

Definition 6. Attribute set of a node i, denoted as A_i. $A_i = \{(\text{type}, \text{cont1}, \text{cont2})\}$. $\forall v \in V$; define A_i as the attribute set of node i, where the type indicates the type of the node i and type $\in \{00, 01, 10\}$, and *cont1* and *cont2* specify the context attributes of the node i (e.g., the start address, the next jump address, etc.). As shown in Figure 4, there are three types of nodes, which are function node, basic block node, and virtual node, explained as follows.

(1) Function node: we monitor noncore functions at the function level and record the calling relationships of functions. The function call relationship can describe the execution path of the program in a coarse-grained manner. For $\forall f \in F(A)$, if there is a direct edge from f_i to f_{i+1}, it means that the function f_i calls the function f_{i+1}. The function node is represented by "00," and the content includes the entry address of the function and the entry address of the called function. The outdegree of a node represents the number of functions directly called by the function represented by the node, and the indegree indicates the number of times the function is called; if a node has no indegree, the node is generally considered to be an entry function. And if a node has no outdegree, the node is generally considered a function call path.

(2) Basic block node: we monitor the core functions at the basic block level. The basic block consists of a section of assembly instructions with only entry and exit. The entry event may be the first statement of the program, the target statement of a conditional jump or unconditional jump, and a conditional jump. The exit event may be a stop statement, a jump statement, or the previous statement of the jump target statement. The basic block node is identified by "01," and the content includes the first address of the basic block, that is, the address of the first instruction in the basic block and the address of the basic block to jump to. The core function is composed of basic block nodes, including start nodes, internal nodes, and exit nodes. Among them, the start node is the function entry block, and the exit node is the end block of the function, such as the ret instruction. The internal node is a basic block divided by the internal instructions of the function.

(3) Virtual node: when a program is running, the loop codes may generate lots of basic block jump edges, which are basically the same, causing a large number of repeated measurements and increasing unnecessary performance costs. This paper introduces a virtual node to represent the loop codes in the function, using the "10" mark, and the content includes the loop entry address, that is, the loop ID and the number of loops.

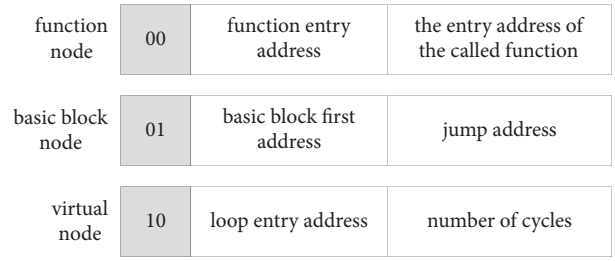


Figure 4: Node structure.

The main function of virtual nodes is to separate common basic blocks and loops to prevent iterative calculations during measurement. The loop structure in the program includes do-while, while, and for. From their assembly code, it can be found that the conditional branch of the backward jump points to the beginning of a loop. This paper uses this common feature as the basis for judging whether to start the loop. When the loop contains jump statements, break, and continue, its control flow will change. For break jump statements, when executed, they jump directly outside the loop, which is equivalent to the exit statement of the second loop, so the normal loop control flow will be recorded, as well as the loop control flow when it encounters a break. When continue is executed, it will jump to the beginning of the next loop, which will also lead to inconsistent control flow, so we will record the number of executions of the continue statement and the control-flow information when the continue is executed. When we encounter a jump statement, we get at least two control-flow metrics, and the repeated codes are only calculated once.

Taking Figure 5 as an example, we explain the control flow measurement and verification as follows.

We can see from Figure 5 that $V = \{F1, F2, B3, B4, B5, B6, B7, B8\}$,

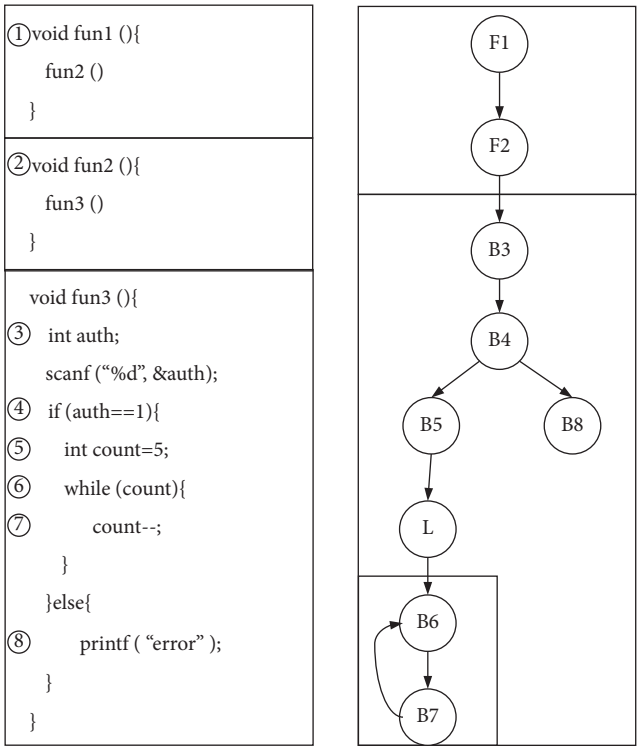


Figure 5: Control-flow graph of a program.

$E = \{F1 \longrightarrow F2, F2 \longrightarrow B3, B3 \longrightarrow B4, B4 \longrightarrow B5, B5 \longrightarrow B6, B6 \longrightarrow B7, B7 \longrightarrow B6, B4 \longrightarrow B8\}$. Suppose *fun1* and *fun2* are noncore functions, and *fun3* is the core function. Node $V_0 \in \{F1, F2\}$ is a function node, and $F1 \longrightarrow F2$ represents *fun1* calling *fun2*. The node $V_1 \in \{B3, B4, B5, B6, B7, B8\}$ is a basic block node. Node L is a virtual node, which points to a loop structure used to store the number of loops. When $auth == 1$, if the virtual node is not introduced, the program execution path is. However, if we introduce the virtual node L, the program execution path will be $P' = \{F1 \longrightarrow F2 \longrightarrow B3 \longrightarrow B4 \longrightarrow B5 \longrightarrow L \longrightarrow B6 \longrightarrow B7\}$. And the program execution path length is reduced from 17 (the length of P) to 8 (the length of P').

As shown in Figure 5, the control-flow graph consists of two paths, and the first path is divided into 3 parts. The measurement results are as follows:

$$\begin{aligned} &F1 \longrightarrow F2\text{: } H_0 = H(H(H(0), F1), F2), \\ &B3 \longrightarrow B5\text{: } H_1 = H(H(H(H(0), B3), B4), B5), \\ &L\text{: } H_2 = H(H(H(0), B6), B7) \| \text{loop\_num}. \end{aligned} \tag{3}$$

Each part and the whole path's measurements are stored in the measurement database in the form of a key-value pair. The key contains the start node and the end node, and the value is the metric value of the part. The first path is stored as $\{\{(F1 \longrightarrow F2\text{: } H_0), (B3 \longrightarrow B5\text{: } H_1), (L\text{: } H_2)\}, H_{\text{path}}\}$.

Then, we show the four types of attacks (in Figure 1) detection with the example program shown in Figure 5:

(1) Branch variable attack: the attack tampers with the verifier's input *auth*, since the control flow that should have jumped to *B5* is changed to jump to *B8*. The control flow leads to an unexpected but legal path.

(2) Loop variable attack: the attack tampers with the value of the loop control variable count, causing the number of while loops to change.

(3) Return address attack: the attack tampers with the return address of the function *fun2*, so that *func2*, which should have returned *fun1*, returns to an illegal address outside of *fun1*, for example, attacker's rogue codes.

(4) Function pointer attack: the attack the attacker tampered with the code pointer to make node B5 turn to B3.

Due to the abovementioned four attacks, the original control flow changes, resulting in a change in the final calculated hash value of the execution path, which can be detected by our method.

6. NSGA-II-Based Granularity-Adaptive Strategy Generation

Many embedded devices (e.g., industrial control devices) have high requirements for real-time performance. In this case, fine-grained control-flow monitoring at the basic block level can provide higher security guarantees, but the performance overhead is relatively high. In contrast, control-flow monitoring at the function level cannot detect Control-Flow Hijacking attacks related to control data due to lack of context information, such as loop variable attacks.

This paper marks the function as a core function or a noncore function and performs basic block-level fine-grained control-flow monitoring for core functions and function-level coarse-grained control-flow monitoring for noncore functions. More marked core functions, more functions need fine-grained control-flow monitoring, but the security benefit improvement may be very little and cause high-performance overhead, especially for resource-limited devices (e.g., IoT or industrial control devices).

We define the optimization goals as security benefits and overhead costs. Then, we transform the above problem into a multiobjective optimization problem as follows:

(1) The goal of optimization:

$$\begin{aligned} &\max\{\text{security}(X)\}, \\ &\min\{\text{overhead}(X)\}. \end{aligned} \tag{4}$$

(2) The condition of constraint:

$$\begin{aligned} &\text{security}(X) > s_{\text{threshold}}, \\ &\text{overhead}(X) < o_{\text{threshold}}. \end{aligned} \tag{5}$$

(3) Decision variables:

$$X = \{x_1, x_2, x_3 \ldots, x_n\}. \tag{6}$$

(4) Parameters:

X is the marking vector, representing the marking scheme on all the functions of a program, showing which functions should be core or noncore functions.

Security(X) refers to the security benefits of the system under the function marking scheme X.

Overhead(X) refers to the overhead costs under function marking scheme X.

$s_{\text{threshold}}$ refers to the value of security benefits when all noncore functions are used.

$o_{\text{threshold}}$ refers to the overhead when all core functions are used.

Suggest there are n functions in a program, for bit i of marking vector X, referred to as x_i. There is

$$x_i = \begin{cases} 1, & \text{core function}, \\ 0, & \text{noncore function}. \end{cases} \tag{7}$$

Among them, the constraint conditions ensure that our function marking scheme will not be all core functions or all noncore functions, resulting in excessive overhead or low-security benefits.

(1) Security benefits: for optimization of a granularity division plan, one of the optimization goals is the security benefit of the system, that is, the security of the control flow of the program because the finer the monitoring granularity is, the more the security of the control flow can be guaranteed. So, we hope to perform fine-grained monitoring on core functions and use the importance of the function and the attacks that can be detected under different granularity monitoring to quantify the security of the control flow as follows:

$$\text{Security} = \sum_{i=1}^{n} \text{dep}_i \times \text{att\_num}_i, \tag{8}$$

$$\text{dep}_i = \frac{\text{fcall}_i}{\text{fcall}_{\text{total}}}. \tag{9}$$

Among them, n is the number of functions in the program and i corresponds to the number of the function.

This paper introduces the concept of function dependency (*dep*) as an indicator to determine whether a function is a core function. Function dependency refers to the proportion of the number of times a function is called to all calling events during the running of the program. The greater the function dependency, the greater the impact on the entire program after the function is destroyed. So, dep_i represents the probability that the i-th function is marked as a core function. $f\text{call}_i$ is the number of calls of the i-th function, and $f\text{call}_{\text{total}}$ is the sum of the number of calls of all functions.

att_num is the number of attack types that can be detected by different granularity monitoring methods and represents the ability of the granularity monitoring method to provide security protection. As shown in the threat model, this paper proposes four control flow-related attacks. The tampering of control data such as code or function pointer and return address tampering can be detected by coarse-grained monitoring methods, and fine-grained monitoring methods can detect all attacks. Therefore, $\text{att\_num}_i \in \{2, 4\}$, and when the i-th function performs coarse-grained monitoring, the value is 2. Otherwise, the value is 4.

(2) Overhead:some resource-limited devices (e.g., industrial control devices) have extremely high requirements for availability. From this perspective, reducing overhead is an important optimization goal. We use the control-flow event to represent a node in the control-flow graph. Especially with fine-grained monitoring, the number of control-flow events is the number of basic block nodes. The total overhead depends on the number of control-flow events.

Our experiments show the relationship between overhead and control-flow events in Figure 6. The abscissa represents the number of control-flow events, and the ordinate represents the overhead. $R^2 = 0.9978 > 0.99$ indicates that it is linearly related. The linear relationship is shown as follows:

$$\text{Overhead} = \sum_{i=1}^{n} \text{overhead} = 0.0044 * \text{cfe\_num}_i + 0.0833. \tag{10}$$

Among them, cfe_num_i is the number of control-flow events of the i-th function. When the function is a core function, the number of control-flow events included is the number of basic block nodes within the function. When the function is a noncore function, the number of control-flow events is 1. The granularity division based on the NSGA-II is described in detail as follows:

Step 1: first, the random population Pt is initialized. Pt is composed of N individuals. Each individual represents a function marking scheme, and each function is represented by a gene. Assuming that there are n functions in the program to be verified, each individual is composed of n genes, and the individual $p = (f_1, f_2, f_3, \ldots, f_n)$, where $f_i \in \{0, 1\}$, $1 \le i \le n$. When $f_i = 0$, it means that the i-th function is marked as a noncore function. And when $f_i = 1$, it means that the i-th function is marked as a core function.

Step 2: the parent Pt generates a child Qt of size N through selection, crossover, and mutation operators and merges the parent and child into a Rt of size $2N$.

Step 3: we perform nondominated sorting on Rt to obtain the nondominated sequence of each function

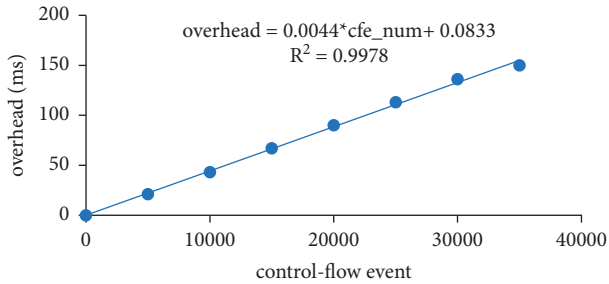


Figure 6: Relationship between overhead and control-flow events.

marking scheme. In this paper, p_{rank} is used to represent the nondominated order of individual p. p_{rank} is determined by the nondominated order of the individual in the entire population. The dominating rule is $\widehat{\text{security}}_p \geq \widehat{\text{security}}_q$ and $\widehat{\text{overhead}}_p \leq \widehat{\text{overhead}}_p$ $p \longrightarrow q$; that is, if the security of plan p is not less than plan q and the cost of plan p is not higher than plan q, it means plan p dominates plan q. We use the min-max standardization method to standardize security and overhead as follows:

$$\widehat{\text{Security}}_p = \frac{\text{security}_p - \min_{\forall p \in P}\text{security}}{\max_{\forall p \in P}\text{security} - \min_{\forall p \in P}\text{security}}, \tag{11}$$

$$\widehat{\text{Overhead}}_p = \frac{\text{overhead}_p - \min_{\forall p \in P}\text{overhead}}{\max_{\forall p \in P}\text{overhead} - \min_{\forall p \in P}\text{overhead}}. \tag{12}$$

Count the dominance set and the number of dominated plans for each plan according to the dominance rules. If the number of dominated plans for plan p is 0, then $p_{\text{rank}} = 0$, and add this plan to the first dominance level $Rank_0$, while continuing to dominate ranking the remaining plans until the entire population is stratified.

Step 4: we sort the congestion degree of Rt and use p_{distance} to represent the congestion degree of plan p.

Step 5: when the sorting rule ($p_{\text{rank}} < q_{\text{rank}}$) or (($p_{\text{rank}} = q_{\text{rank}}$) and ($p_{\text{distance}} < q_{\text{distance}}$)) is satisfied, it means that plan p is better than plan q. We select the first N better individuals to form a new generation population P_{t+1} for the next iteration.

Step 6: when judging whether the predetermined number of iterations G is reached, output the first-ranked optimal function marking scheme if it is reached; otherwise, proceed to Step 2.

Table 1 describes the optimization process.

7. Runtime Control-Flow Attestation Protection for Control-Flow Measurements Based on SGX

Intel SGX [17, 18] (Intel Software Guard Extensions) is an extension of the Intel CPU, which encapsulates the programs that need to be protected in the enclave. All privileged or nonprivileged software cannot access the content in the enclave and can be used to protect applications. The key codes and data are not tampered with by malicious software or high-level system management software (such as OS, VMM, and BIOS). The root of trust of SGX only includes the CPU, which greatly improves the system's security. Therefore, this paper uses Intel SGX to provide a trusted execution environment for the measurement of the program.

As shown in Figure 7, the prover is divided into safe area and unsafe area. The runtime tracking part of the program is placed in the unsafe area, and the measurement and attestation part are placed in the safe area (SGX enclave), ensuring the secure storage of the attestation report and secret key information.

When the prover receives the verification request from the verifier, it runs the program to be verified according to the specified input. When the function call is executed, it will jump to the coarse-grained interceptor; when the jump instruction inside the core function is executed, it will jump to the fine-grained interceptor and then send the intercepted address information to the node inspection module. Nodes are distinguished and then sent to the measurement module to generate the accumulated hash value of each block. The attestation module uses enclave's private key to sign the measurement result and random number, generates an attestation report, and sends it to the verifier.

All key modules are introduced in the following.

7.1. Runtime Tracing. This part is used to track the verified application. This paper rewrites the pin tool in Intel Pin-3.15 and makes it the tool for detecting control-flow events when the program is running. In order to obtain the function call relationship in the program, the program needs to be instrumented at the function level. The instrumentation points are mainly in the call and ret instructions. In order to obtain the internal control-flow information of the core function, it is necessary to perform basic block-level instrumentation to monitor indirect jump statements.

7.2. Coarse-Grained Interceptor. During program execution, when a function call instruction is encountered, GACFA will transfer the program control flow to the coarse-grained interceptor. The interceptor obtains the address of the function at this time, the address of the called function, and the node type of what we insert. Then, through the SGX *ECALL* instruction, it switches to the SGX enclave, passes the acquired address information to the node inspection module in the safe area, and then returns the program control flow to the program execution.

7.3. Fine-Grained Interceptor. When the core function of the program is executed, the fine-grained interceptor intercepts branch information, obtains the first address of the basic block and the node ID we inserted, and sends the information to the node inspection module in the security zone.

Table 1: Granularity division algorithm based on NSGA-II.

```
Input: F(A), A.exe, A_inputs[ ]
//The function collection of the program, binary file, and input list
Output: Res
//Output optimal granularity division plan
{
   //F(A) is the set of all functions of the program
   F(A).count = 0; //the number of functions of the program A
   F(A).size[ ] = 0; //the number of control-flow events of each function
   static_func (F(A), A.exe, inputs[ ], F(A).count , F(A).size[ ]);
   //Count the number of function calls and control-flow events in each function
   GranularityDivide (F(A).count , F(A).size[ ], Res);
   //Granularity division, return the optimal granularity division plan
}
GranularityDivide (count, size[ ], best_scheme)
{
   g = 0;
   InitPopulation (Pt: N);
   //The population Pt is initialized, containing N individuals representing a granularity division plan each
   while g ≤ G do:
      CreateOffspring (Pt, Qt);
      //The crossover and mutation on Pt and Qt produce offspring
      Rt = Merge (Pt, Qt);
      for p in Rt do:
         //Objective calculation
            security = f_security (count, size[ ], p);
         //Calculate the security benefits of each individual p according to formulas (8) and (9)
         overhead = f_overhead (count, size[ ], p);
         //Calculate the overhead costs of each individual p according to formula (10)
      end for
      NonDominatedSorting(Rt);
      //Non-dominated sorting based on objective function according to formulas (11) and (12)
      CrowdingDistanceSorting(Rt);
      //Sort the congestion degree of Rt
      Pt = Rt[0, N];
      //Select the top N better solutions
      g = g + 1;//go to the next generation, trying to get better granularity division plan
      end while
      best_scheme = Pt[0];
      //Return to the optimal partition plan
}
```

7.4. Node Check Module. This module judges the node type at this time. If the type changes, it initializes the measurement result and restarts accumulating hash. If a virtual node is detected, the number of cycles needs to be recorded and sent to the measurement module.

7.5. Measurement Module. In order to ensure the security of the measurement process, we perform the SHA256 hash calculation in the enclave container and send the metric value to the attestation module.

7.6. Attestation Module. Use the SGX function to create public and private key pair required for signing the random number and measurement to generate the attestation report and send the attestation report to the verifier.

8. Evaluation

The following is an evaluation and analysis of our proposal GACFA from three aspects: functionality, overhead, and security.

8.1. Functional Evaluation. We use the SNU real-time benchmark [24] to test the effectiveness of GACFA, which is dedicated to the performance evaluation of C/C++ programs. GACFA aims to achieve a combination of coarse- and fine-grained attestation methods, so we use multi-objective optimization algorithms for function marking to determine the Control-Flow Attestation scheme.

8.1.1. Multiobjective Optimization of NSGA-II. In order to prove the effect of NSGA-II optimization, we selected the program "*adpcm-test*" for the analysis, which contains the

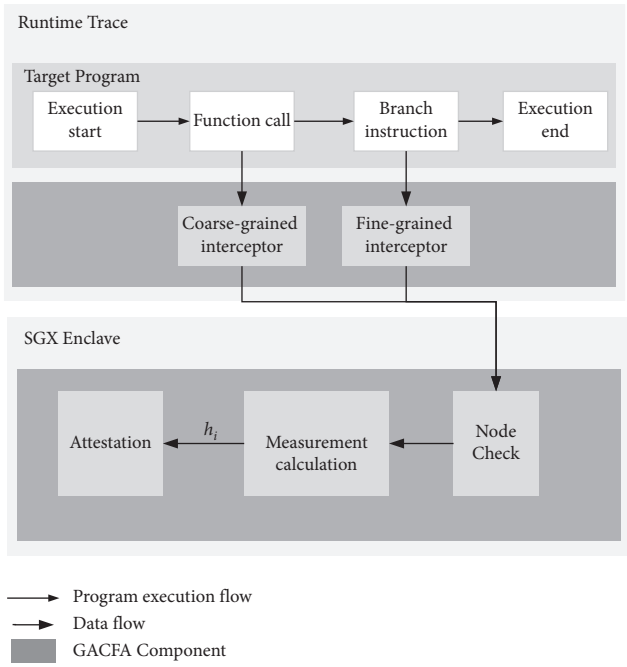


Figure 7: Runtime protection on program-end (the prover) based on the SGX.

most functions (15 functions) in the SNU test set. The program's information and test parameters are listed in Table 2. We get the optimal function marking scheme according to the algorithm described in Section 6. The result is shown in the last column of Table 2, in which functions marked as "1" are chosen to be measured in the granularity of the basic block, while functions marked as "0" are measured as one piece.

Figure 8 shows the optimizing marking process of program "*adpcm-test*." The blue/red line shows the change trend of the security/overhead goal of each generation (with different granularity division schemes). As we can see, generation 67 is the optimal solution after convergence with "010100110000111" as its gene (shown as the last column in Table 2).

8.1.2. Control-Flow Attestation. The verifier sends an attestation request to the prover (in our case, an embedded industrial control computer) to verify the security of the industrial control program. Take the program simulating user verification process, described in Figure 5 in Section 5, as an example. The program includes two paths. When the input is 1, it is an authorized path, and with the other input, it is an unauthorized path.

The verifier can perform measurement on both paths in advance to get the expected measurement values shown in Figure 9.

Figure 10 shows a successful attestation process. As Figure 10(a) shows, after receiving the attestation request, the control-flow measurements are gotten from the running program and the report and signature are generated and sent to the verifier-end. As Figure 10(b)

Table 2: Parameters and optimal function marking solution of program "*adpcm-test*."

Function name	Number of calls	Number of dangerous function calls	Number of control-flow events	Optimized marking solution
Abs	0	0	5	0
logsch	2000	4	7	1
invqah	0	0	2	0
Uppol1	4000	2	10	1
Uppol2	4000	2	13	0
upzero	4000	3	11	0
scalel	4000	2	2	1
logscl	2000	5	7	1
invqxl	0	0	2	0
encode	1000	2	37	0
decode	1000	1	24	0
reset	1	0	10	0
fabs	15729	4	5	1
filtez	4000	4	4	1
filtep	4000	5	2	1

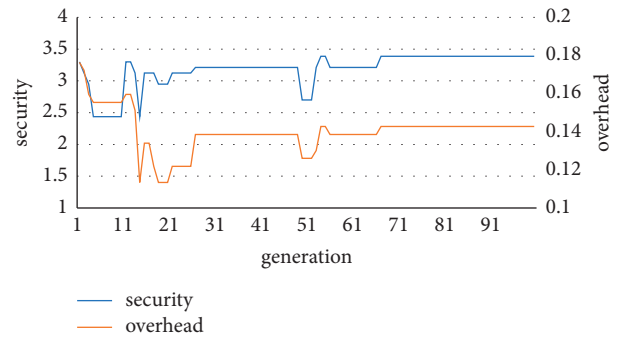


Figure 8: The optimizing marking process of program "*adpcm-test*".

indicates, the verifier verifies the data sent by the program (the prover-end) with the measurement database. Since there is no Control-Flow Hijacking attack, the attestation is successful.

8.1.3. Attack Detection. We launch all 4 kinds of CFH attacks (Attacks 1–4) described in Section 3 to the program "*adpcm-test*" (containing 15 functions) and test three different detection methods for comparative attack detection experiments, namely, Fine-Grained Control-Flow Attestation [6] (noted as FGCFA, with the granularity of basic block), GACFA (our proposal, with the optimal function marking solution shown in Table 1), and Coarse-Grained Control-Flow Attestation [25] (noted as CGCFA, with the granularity of the function).

In the real world, the vulnerability may be found in any function. Therefore, we use the potential threat ratio of 1 : 1 (7 functions without attack and 8 functions with attacks), 1 : 2 (5 functions without attack and 10 functions with attacks), and 1 : 4 (3 functions without attack and 12 functions with attacks) to represent the ratio of the functions without

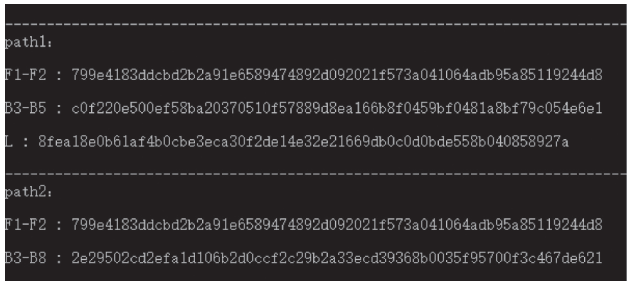


Figure 9: Expected measurements of two execution paths.

vulnerability to the functions with vulnerability. That is, the function potential threat ratio is 53%, 67%, and 80%. We conducted 3 rounds of random attack tests on the program, one round for each ratio. The average detection rate results are shown in Figure 11.

In Figure 11, the FGCFA can detect all attacks, so the detection rate is 1 under different ratios. The CGCFA detection capability is related to the simulated attack category, but it does not consider the risk function is more vulnerable to attack, so the detection rate under different proportions is low. With the increase in the number of attacks on threat functions, the detection capability of the GACFA solution increases. At the potential threat ratio of 53.3%, the detection rate is 0.7, which is 30% lower than that of the FGCFA solution and is 2.5 times the CGCFA solution. The detection rate of this scheme is 24% lower than that of the FGCFA solution and 2.69 times the CGCFA solution. Therefore, as the probability of being attacked by threat functions increases, the detection capability of the scheme in this paper gradually approaches the FGCFA solution, but the performance overhead remains unchanged.

8.1.4. Runtime Control-Flow Attestation Protection with the SGX. In this section, we discuss the security of GACFA. The security requirement of GACFA is to verify the integrity of the control flow of the program to be verified on the prover to ensure that the measurement results are not tampered with.

The online attestation module includes the program's runtime tracking component and measurement attestation component. The runtime tracking component is a binary file after the pin tool has been instrumented. Like C-FLAT [6], we believe that the static measurement includes the target program and runtime tracking, and the integrity of this part will not be tampered with. GACFA places the measurement and authentication components of control-flow information in the SGX enclave and uses the program isolation mechanism of the SGX enclave to ensure the validity of the authentication report.

In order to verify the security of the SGX enclave protected measurement and attestation at the program-end, this paper attempts to tamper with the control-flow hash value not protected and protected by the SGX enclave. The measured program is still the one shown in Figure 5. Figure 12(a) shows the accumulated hash calculation process of path 1 of the program. Then, we shut down the SGX and tamper measurement code from memory. Figure 12(b) shows the 3rd part's hash value has changed, and the tampering is successful. However, after placing the measurement program in the SGX enclave, trying to access measurement code outside the enclave is prohibited, and the tampering on final measurement failed, which is shown in Figure 12(c).

According to Figure 12, we can find that by introducing SGX, GACFA can reliably verify the integrity of the control flow of the program and ensure that the measurement results and attestation report from being tampered with.

8.2. Performance Analysis. Offline program analysis is performed by the verifier and does not affect the execution process of the program, so we only consider the performance overhead of the online attestation phase. The runtime program attestation time includes the control-flow information collection time t_col, the measurement time t_mea, and the signing time t_sig, so the total runtime attestation time is shown as follows:

$$t_{\text{total}} = t_{\text{col}} + t_{\text{mea}} + t_{\text{sig}}. \tag{13}$$

We have implemented the FGCFA, GACFA, and CGCFA for the *adpcm-test* program. In particular, for our proposal GACFA, we perform instrumentation according to the optimal function marking scheme 010100110000111 given in Section 8.1, of which 8 functions perform coarse-grained instrumentation and 7 functions perform fine-grained instrumentation. Other procedures, such as measurement and attestation, in all three methods are the same.

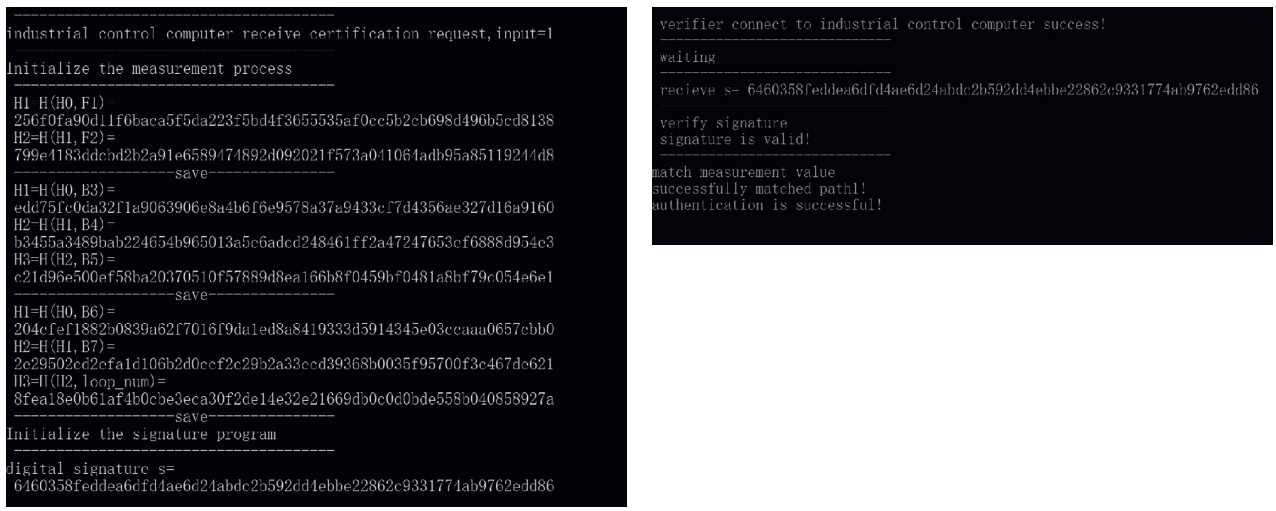
Figure 13 shows the time of online attestation for the three methods. Although our time cost is increased by 31.79% compared with the CGCFA, which cannot detect any basic block-level CFH attacks, our time cost is reduced by 56.99% compared with FGCFA. This is because the necessary time of control-flow collection and measurement is greatly reduced compared to the FGCFA.

Figure 14 shows the comparison of the measurement time for several different programs in the SNU data set. Although the average size of SNU programs is small, we can still see that, for different programs, the measurement time cost of our proposal is much lower than the FGCFA.

8.3. Balance of Security and Overhead. In order to evaluate the security and overhead balance capability of different Control-Flow Attestation schemes, we propose the concept of security/performance benefit ratio for different schemes as $\text{spbr}_{x/y}$ (x, y are different schemes), that is, the detection rate change ratio/detection time change ratio. Apparently, when $\text{spbr}_{x/y}$ is bigger, scheme x is better than y. The calculation method of $\text{spbr}_{x/y}$ is shown as follows:

$$\text{spbr}_{x/y} = \frac{\text{detection}_{\text{ratex}} - \text{detection}_{\text{ratey}}/\text{detection}_{\text{ratey}}}{\text{detection}_{\text{timex}} - \text{detection}_{\text{timey}}/\text{detection\_time}_{y}}. \tag{14}$$

$\text{spbr}_{\text{FGCFA/GACFA}}$ and $\text{spbr}_{\text{GACFA/CGCFA}}$ are shown in Table 3. It can be seen from the table that $\text{spbr}_{\text{GACFA/CGCFA}}$ is



(a) (b)

Figure 10: The prover-end and the verifier-end results in attestation: (a) the prover-end results and (b) the verifier-end results with no control-flow tampering attack.

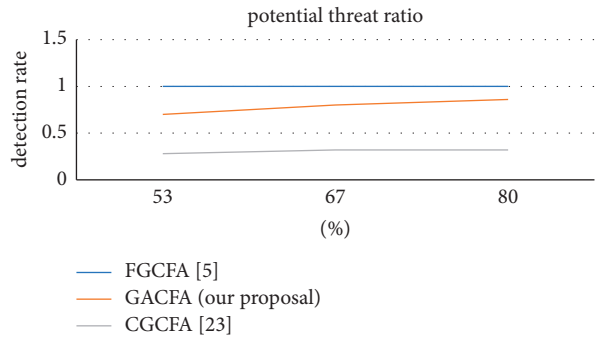


Figure 11: Comparison of attack detection capabilities under different attack scenarios.

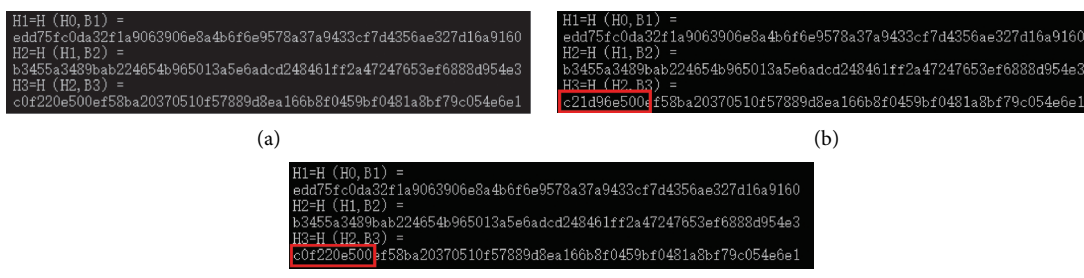


Figure 12: SGX antitampering results. (a) Expected path and final hash result (*H3*). (b) Tampering (*H3)* succeeded. (c) Tampering (*H3*) failed.

the highest when the CGCFA is changed to our scheme (GACFA), and $spbr_{GACFA/CGCFA}$ is 13.7, 25.1, and 43.0 times that of $spbr_{FGCFA/GACFA}$ in the case of three different potential threat ratios (specified in Section 8.1). This means changing from CGCFA to our scheme can achieve a much better balance of security and overhead. That is, higher security can be obtained with less performance overhead in our scheme.

9. Discussion

We propose GACFA for CFH detection of embedded programs to detect control data and noncontrol data attacks that indirectly affect control flow. To ensure the security of attestation and measurement, we put the modules into the SGX enclave, so that measurement calculation and attestation report signing and generation will not be tampered

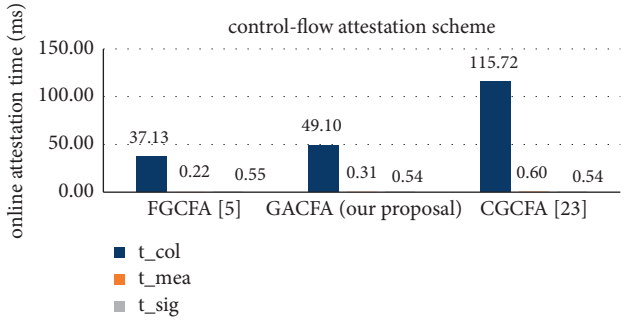


Figure 13: Comparison of online attestation time on different schemes.

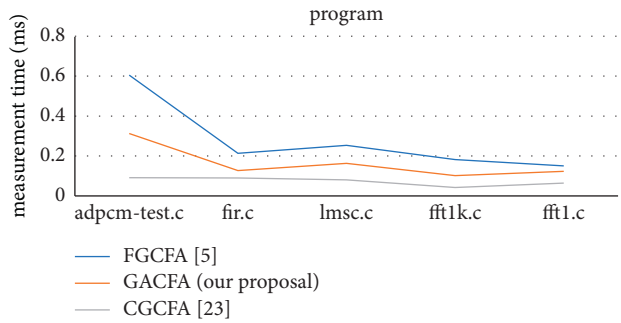


Figure 14: Comparison of measurement time on different schemes with different programs.

Table 3: Security/performance balance capability comparison in different threat scenarios.

Potential threat ratio (%)	$spbr_{FGCFA/GACFA}$	$spbr_{GACFA/CGCFA}$
53	0.32	4.69
67	0.18	4.69
80	0.12	5.27

with by privileged attackers. However, the SGX is proposed to protect applications. But the Intel pin tools are used to take care of the process of monitoring and extracting control flows of the running program, which cannot be protected by the SGX because the tools rely on the kernel. This means if the pin tools are tampered with, the control-flow information we get and use in the SGX enclave cannot be trusted. Hardware-based method, such as LO-FAT [16], ATRIUM [7], seems to be promising to solve this problem. LO-FAT [16] extends branch tracking of processor pipeline with additional logic to achieve efficient tracing of control-flow information, along with measurement calculation and attestation, and implement the proof-of-concept system based on a RISC-V SoC. ATRIUM [7] is tightly integrated with a processor and can extract the executed instructions and memory addresses and generate the final measurement and attestation report. Of course, a hardware-based method means extra hardware support and costs.

Besides, our proposal can only detect CFH attacks based on program execution-related data (e.g., jump address, branch, and loop variables), not pure data attacks (data-oriented programming attacks, DOP attacks) [26]. The DOP attacks manipulate data in the program and tamper with key variables but not control data and noncontrol data mentioned above, which means the current control-flow detection method may be bypassed. Hu et al. [27] proposed several new solutions, such as data flow integrity protection and fine data flow randomization. However, these solutions on data flow cost much more than CFI methods and need more research before they are used in practice.

10. Conclusions

Control-Flow Hijacking attacks have become the main method of exploiting vulnerabilities, seriously threatening the security of industrial control systems. This paper proposes a NSGA-II based Granularity-Adaptive Control-Flow Attestation (GACFA) for CFH detection of embedded programs, which can be used to detect control data attacks and noncontrol data attacks that indirectly affect control flow. A control-flow representation model is used to reduce the complexity of programs' control-flow graph, while the NSGA-II algorithm is used in offline analysis to optimize the granularity division strategy to maximize system security benefits and minimize overhead. Besides, runtime

protection for the GACFA at the program-end with SGX is proposed to protect the integrity and confidentiality of control-flow measurement data. Experiments have shown that the algorithm can obtain a relatively balanced coarse and fine granularity between control-flow security and runtime efficiency. In addition, we evaluated the performance and security of the runtime module and found that compared with the current fine-grained Control-Flow Attestation method, the security/performance benefit of adopting our proposal over CGCFA is 13.7, 25.1, and 43.0 times that of adopting FGCFA over ours in different threat scenarios.

Data Availability

All data are available from the corresponding author upon request.

Conflicts of Interest

The authors have no conflicts of interest.

Acknowledgments

This work was supported by grants from the National Key Research and Development Program of China (Grant no. 2016YFB0800204) and the Open Research Fund of Beijing Key Laboratory of Trusted Computing.

References

[1] H. Ke, H. Wu, and Y. Dongmei, "Towards evolving security requirements of industrial internet: a layered security architecture solution based on data transfer techniques,"pp. 504–511, Association for Computing Machinery, Beijing China, December 2020.

[2] N. Jr, T. Fraser, J. Molina, and W. A. Arbaugh, "Copilot - a coprocessor-based kernel runtime integrity monitor," in *Proceedings of the 13th USENIX Security Symposium*, pp. 179–194, USENIX Association, San Jose, CA, USA, August 2004.

[3] A. Seshadri, A. Perrig, L. Doorn, and P. Khosla, "SWATT: SoftWare-based ATTestation for embedded devices," in *Proceedings of the IEEE Symposium On Security And Privacy*, pp. 272–282, IEEE, Berkeley, CA, USA, May 2004.

[4] Z. Chao, W. Tao, and Z. Chen, "Practical control flow integrity and randomization for binary executables," in *Proceedings of the 2013 IEEE Symposium on Security and Privacy (SP)*, pp. 559–573, IEEE, Berkeley, CA, USA, May 2013.

[5] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross, "Control-flow bending: on the effectiveness of control-flow integrity," in *Proceedings of the 24th USENIX Security Symposium*, pp. 161–176, USENIX, Berkeley, CA, USA, August 2015.

[6] T. Abera, N. Asokan, and L. Davi, "C-flat: control-flow attestation for embedded systems software," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security ACM*, pp. 743–754, ACM, Darmstadt, Germany, October 2016.

[7] S. Zeitouni, G. Dessouky, O. Arias, D. Sullivan, and A. R. Sadeghi, "Atrium: runtime attestation resilient under memory attacks," in *Proceedings of the 2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 384–391, IEEE, Irvine, CA, USA, November 2017.

[8] X. Ge, W. Cui, and T. Jaeger, "Guarding control flows using intel processor trace," in *Proceedings of the the Twenty-Second International Conference*, pp. 585–598, ACM, Redmond, WA, USA, Apr 2017.

[9] P. F. Qiu, Y. Q. Lyu, J. Zhang, D. Wang, and G. Qu, "Control flow integrity based on lightweight encryption architecture," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, pp. 1358–1369, 2017.

[10] N. Burow, S. A. Carr, J. Nash et al., "Control-flow integrity," *ACM Computing Surveys*, vol. 50, no. 1, pp. 1–33, 2017.

[11] R. Ding, C. Qian, C. Song, B. Harris, and W. Lee, "Efficient protection of path-sensitive control security," in *Proceedings of the 26th USENIX Security Symposium*, pp. 131–148, USENIX, Vancouver, BC, Canada, August 2017.

[12] D. Jung, M. Kim, J. Jang, and B. B. Kang, "Value-based constraint control flow integrity," *IEEE Access*, vol. 8, Article ID 50542, 2020.

[13] MR. Khandaker, W. Liu, A. Naser, Z. Wang, and J. Yang, "Origin-sensitive control flow integrity," in *Proceedings of the 28th USENIX Security Symposium*, pp. 195–211, USENIX, Santa Clara, CA, USA, August 2019.

[14] M. Khandaker, A. Naser, W. Liu, Z. Wang, Y. Zhou, and Y. Cheng, "Adaptive call-site sensitive control flow integrity," in *Proceedings of the 2019 IEEE European Symposium on Security and Privacy (EuroSP)*, pp. 95–110, IEEE, Stockholm, Sweden, June 2019.

[15] H. Jang, M. Park, and H. Dong, "IBV-CFI: efficient fine-grained control-flow integrity preserving CFG precision," *Computers & Security*, vol. 94, 2020.

[16] G. Dessouky, S. Zeitouni, T. Nyman et al., "LO-FAT: low-overhead control flow ATtestation in hardware," in *Proceedings of the 54th Annual Design Automation Conference 2017*, pp. 1–6, IEEE, Austin, TX, USA, June 2017.

[17] I. Corporation, *Intelsoftware Guard Extensions Programming Reference*, Intel, San Jose, CA, USA, 2014.

[18] I. Anati, S. Gueron, S. Johnson, and V. Scarlata, "Innovative technology for CPU based attestation and sealing," in *Proceedings of the Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, pp. 1–6, Tel-Aviv, Israel, June 2013.

[19] F. Schuster, M. Costa, C. Fournet et al., "VC3: trustworthy data analytics in the cloud using SGX," in *Proceedings of the IEEE Symposium on Security and Privacy SP*, pp. 38–54, San Jose, CA, USA, 2015.

[20] F. Shaon, M. Kantarcioglu, Z. Lin, and K. Latifur, "SGX-BigMatrix: a practical encrypted data analytic framework with trusted processors," in *Proceedings of the 2017 ACM SIGSAC Conference*, pp. 1211–1228, ACM, Dallas, TX, USA, October 2017.

[21] J. Wang, H. Zhi, and Y. Zhang, "Enabling security-enhanced attestation with intel SGX for remote terminal and IoT," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, pp. 88–96, 2017.

[22] V. Karande, E. Bauman, Z. Lin, and K. Latifur, "SGX-log: securing system logs with SGX," in *Proceedings of the ACM Asia Conference on Computer and Communications Security (ASIA CCS)*, pp. 19–30, ACM, New York, NY, United States, April 2017.

[23] C. Khammassi and S. Krichen, "A nsga2-lr wrapper approach for feature selection in network intrusion detection," *Computer Networks*, vol. 172, 2020.

[24] "SNU real-time benchmarks," http://www.cprover.org/satabs/examples/SNU_Real_Time_Benchmarks/.

[25] Y. Yu, P. Wang, Y. Zhang, H. Zhang, and H. Zhang, "Detection of control flow attacks based on return address signature," *Journal of East China University of Science and Technology*, vol. 46, pp. 800–806, 2020.

[26] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer, "Non-control-data attacks are realistic threats," in *Proceedings of the 14thUnisex Security Symposium*, Baltimore, MD, USA, July 2005.

[27] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang, "Data-oriented programming: on the expressiveness of non-control data attacks," in *Proceedings of the 37th IEEE Symposium Gainurity and Privacy*, pp. 969–986, IEEE, San Jose, CA, USA, May 2016.