Control Flow Checking at Virtual Edges

LiPing Liu[1*], LinLin Ci[1], Wei Liu[1], Hui Yang[1]
[1] Computer department, Beijing Institute of Technology
Beijing China
[e-mail: liuliping_bit@163.com, cilinlin_bit@126.com, cilinlin_bit@126.com, zlj-1943@163.com]
*Corresponding author: LiPing Liu

Received October 2, 2016; revised May 8, 2016; accepted November 29, 2016; published January 31, 2017

Abstract

Dynamically checking the integrity of software at run-time is always a hot and difficult spot for trusted computing. Control-flow integrity is a basic and important safety property of software integrity. Many classic and emerging security attacks who introduce illegal control-flow to applications can cause unpredictable behaviors of computer-based systems. In this paper, we present a software-based approach to checking violation of control flow integrity at run-time. This paper proposes a high-performance and low-overhead software control flow checking solution, control flow checking at virtual edges (CFCVE). CFCVE assigns a unique signature to each basic block and then inserts a virtual vertex into each edge at compile time. This together with insertion of signature updating instructions and checking instructions into corresponding vertexes and virtual vertexes. Control flow faults can be detected by comparing the run-time signature with the saved one at compile time. Our experimental results show that CFCVE incurs only 10.61% performance overhead on average for several C benchmark programs and the average undetected error rate is only 9.29%. Compared with previous techniques, CFCVE has the characteristics of both high fault coverage and low memory and performance overhead.

Keywords: Trusted computing, Dynamic measurement, Control-flow errors, Error detection, Virtual edges.

1. Introduction

Distributed computing and worldwide business transactions over open networks, such as the Internet, increasingly demand for secure communication and secure operation due to rising online fraud and software attacks [1]. Some of these vulnerabilities are due to the complexity and architectural constraints of the underlying execution environment (CPU hardware and commodity operating systems), some are due to poor software development practices and lack of software security in applications [2]. In this context, the integrity of system software and applications is a fundamental requirement and necessary consequence in order to ensure trust in the computing infrastructure [3].

Trusted Computing as proposed by the Trusted Computing Group (TCG) offers a technology that is able to verify the integrity of executable content through remote attestation. The cores of trusted computing technology are trusted computing base and trusted chain [4, 5], and trusted measurement is a key problem of this technology [6, 7]. Trusted computing treats the integrity as a fundamental attribute of trust. However, such attestation mechanisms provide only integrity verification at load-time but not at run-time: An attacker can change the flow of execution of a program, e.g., via buffer overflow attacks that are despite numerous counter measures still a great security concern in software systems today.

Some attacks do not need to inject new code, but instead use code that already exists in the process's memory. Existing protection mechanisms such as marking the stack as non-executable cannot detect this class of attacks because only instructions are executed that reside in valid code pages. Moreover, the new attacks generalize the original return-into-libc attack by allowing the attacker arbitrary computation without calling any functions [8]. In a traditional return-into libc attack, an attacker could execute only straight-line code without using branching, and could only invoke functions that reside in libc. In the new attacks, an attacker overwrites the stack with return addresses that point to existing code fragments in the program or system libraries.

SEU-induced soft errors have been known as one of the major threats to functionality and reliability of space-borne computers and their host spacecrafts. Soft errors may be explicit bit flips in latches or memories, or glitches in combinational logics that can propagate and be captured in latches [9]. If not handled properly, such errors can cause illegal accesses to peripherals, memory overflow, data corruption, false and sometimes fatal data or action outputs, and so on. Therefore, it is necessary to detect and correct errors in control flows hopefully before damages are caused.

Various control-flow checking techniques have been proposed in the literature. These techniques are suggested in literature that would fall into two general classes, hardware [10, 11, and 12] or software [13, 14, 15, 16, and 17] redundancy. The methods based on hardware redundancy have a better fault coverage but need additional hardware or modification of the existing hardware and cannot guarantee portability to various platforms. Software-based techniques have less fault coverage and larger delay; however, mean lower cost and overhead on the system and can be utilized in different types of industrial systems due to their flexibility. The basic idea of software control flow checking is to partition the program into basic blocks (branch-free parts of code) [14]. For each block a deterministic signature is calculated and saved somewhere during compile time; then errors can be detected by comparing the run-time signature with the saved one. This method does not require any additional hardware and can be automatically applied for software control flow checking. It should be noted that, in practical industrial applications, whatever approach is used, whether it is software-based or hardware-based, it should be able to handle the errors mentioned above as much as possible and, in doing so, impose as little memory overhead and as little increase in execution time as possible.

We propose a control flow checking technology based on updating a signature at the directed edges called CFCVE, which assigns a unique signature occupying the least bit of each basic block and updating the signatures at the virtual edges of the control flow graph, allowing it to detect all the single inter-block control flow error. CFCVE is comparable in performance overhead incurred with CFCSS. However, its fault coverage is always higher than CFCSS and is comparable with RSCFC. Furthermore it overcomes the limitations of CFCSS with less memory than RSCFC. This solution is inspired by [13, 14, 15, 16, and 17] and incorporates their advantages. The contributions of this work are as follows:

- Our work is the first to update the signatures at the directed edges instead of at the nodes of control flow graph.
- We propose a novel fitness factor in this paper that can compare different approaches, based on their fault coverage, memory overhead，performance overhead and error detection latency.
- We assess control flow checking technology based on updating a signature at the directed edges under novel fitness factor.

The remainder of the paper is organized as follows. In Section 2 the related works on software control flow checking methods are reviewed, while Section 3 describes the proposed approach. In Section 4 we introduce how to further enhance the error detection capability of the proposed method. The capabilities of the proposed technique are analyzed in Section 5. Section 6 reports the experimental results we gathered and finally section 7 draws some conclusions.

2. Related Work

A variety of defense mechanisms are proposed to detect and correct control flow errors (e.g., [10, 11, 12, 13, 14, 15, 16, 17, 18, 19, and 20]). Clearly, this is a crowded, important research area. Next we elaborate on some of the pieces of work most closely related to ours.

The most important three software-based solutions proposed in the literature are the techniques called Enhanced Control Flow Checking Using Assertions (ECCA) [13] and Control Flow Checking by Software Signatures (CFCSS) [15] and On-line control flow error detection using relationship signatures among basic blocks (RSCFC) [14].

ECCA, firstly, assigns a unique prime number identifier (BID) to each basic block of a program; then a test assertion and a set assignment, which are composed of a global integer variable (id) and the BID, are individually inserted into the beginning and the end of each basic block. During the execution, the id is dynamically updated and it can transfer a control flow error into a divide by zero error. ECCA is able to detect all the single inter-block control flow error, except the faults that cause an incorrect decision on a conditional branch. Due to its complexity of the test and set assertions, ECCA has higher memory and time overhead than CFCSS.

CFCSS assigns a unique signature s_i to each basic block and uses a global variable (G) to contain the run-time signature. When control transfers from one basic block to another, CFCSS uses the first instruction to compute the signatures of the destination block from the signature of the source block and a pre-computed variable d, which is the EXOR difference between the signatures of the source and destination blocks, and updates G with the value at the beginning of the basic block; in the following instruction, CFCSS then tests whether G is equal to the destination block's signature. If the control can enter from multiple blocks, then an adjusting signature (D) is assigned in each source block and used in the destination block to compute the signature. In absence of errors, the signature contained in G is equal to the signature of destination block. However, CFCSS cannot cover control flow errors if multiple source blocks share multiple branch-fan-in blocks as their destination blocks, aliasing could occur between legal and illegal branches.

One of the works in this field is Relationship Signature CFC (RSCFC) in which the program is divided into some basic blocks. In the first stage, the relationship between blocks is extracted and then based on the kind of the relationship, a signature is assigned to each block in which the existing relationships are coded in it. The faults in the control flow of the program are detected by logic AND the run-time signatures with the information at the beginning and end of the blocks. In comparison to the previous works, this method has more fault coverage and a better efficiency. However, RSCFC, codes the transfer relationship among basic blocks into each block's signature, which leads to the signature of each block have many bits exceeding the limitation of machine word possibly, consumes more memory than CFCSS. Although this situation can be cooperated by grouping basic blocks, however, error detection latency will be increased.

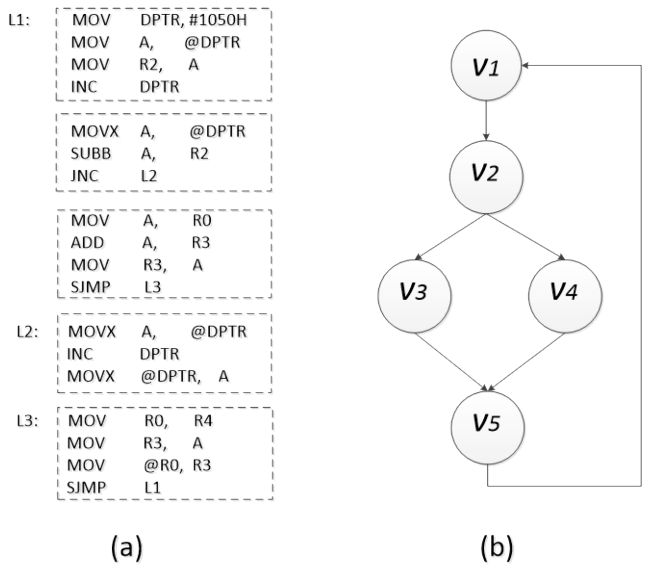
3. Methodology

To start with, definitions of relevant concepts are presented in table 1 for the sake of clarity. CFCVE checks the control flow of the program using a dedicated register called the global signature register (GSR), which contains the run-time signature G associated with the current node (the node that contains the instruction currently executed) in the program flow graph. Every basic block (represented by a node v_i in the program flow graph) is identified and assigned a unique signature s_i when the program is compiled. Let G_i be the run-time value of G when the program flow is at node v_i. Under normal execution of the program (no errors), G_i should be equal to s_i. If G contains a number different from the signature associated with the current node, it means an error has occurred in the program [13].

Table 1. Definitions of relevant concepts used in this paper

Concept	Definition
Basic Block	A Basic Block (BB) is a maximal set of ordered non-branching instructions (except in the last instruction) or branch destinations (except in the first instruction) in which the execution always enters at the first instruction and leaves via the last instruction [13].
Control Flow Graph	A program P can be represented with a directed graph composed of a set of node V and a set of edges E,$P=\{V,E\}$, where $V=\{v_1, v_2,\dots,v_i,\dots,v_m\}$ and $E=\{e_1,e_2,\dots,e_i,\dots,e_m\}$ Each node v_i represents a basic block and each edge e_i represents the branch $br_{i,j}$ from v_i to v_j. This directed graph is called the control flow graph (CFG).
Vertex	Basic blocks are denoted by vertexes $v_i(i\in\{1,2,\dots,N\})$ in the control flow graph CFG, where N is the total number of basic blocks. $V:\{v_i;i\in\{1,2,\dots,N\}\}$, a set of vertexes denoting basic blocks.

$suc(v_i)$	A set of successors of v_i in CFG, if and only if $br_{i,j} \in E$, then the node $v_j \in suc(v_i)$.
$pred(v_i)$	A set of predecessors of v_i in CFG, only if $br_{j,i} \in E$, then the node $v_j \in pred(v_i)$.
Virtual Vertex	A Virtual Vertex (VV) denotes a basic block that does not contain any other effective instructions except an unconditional jump instruction and has only one successor vertex and one predecessor vertex. $vv_{i,j}$ represents a virtual vertex whose predecessor vertex is vertex v_i and successor vertex is v_j.
Edge	A legal branch from one basic block to another is denoted by a directed edge between the two correspondent vertexes in CFG.
Virtual Edge	A Virtual Edge (VE) is an Edge in which a Virtual Vertex is embedded.
Control-flow error	A control-flow error (CFE) is said to have occurred if the sequence of instructions executed in presence of a fault is different from the fault-free sequence.
Internode CFEs	Internode CFEs occur if the program control before and after the illegal jump resides in different nodes.
Intranode CFEs	Intranode CFEs occur if the program control before and after the illegal jump resides in the same node. [22]
XOR operation	The xor-difference of a and b is the result of performing the bitwise XOR operation ($\oplus$) of a and b, i.e., xor-difference = a $\oplus$ b, where a and b are binary numbers.


```
L1:  MOV    DPTR, #1050H
     MOV    A,    @DPTR
     MOV    R2,   A
     INC    DPTR

     MOVX   A,    @DPTR
     SUBB   A,    R2
     JNC    L2

     MOV    A,    R0
     ADD    A,    R3
     MOV    R3,   A
     SJMP   L3

L2:  MOVX   A,    @DPTR
     INC    DPTR
     MOVX   @DPTR,  A

L3:  MOV    R0,   R4
     MOV    R3,   A
     MOV    @R0,  R3
     SJMP   L1
```

Fig. 1. A typical example of program and its CFG

The structure of a program can be represented by a CFG, where nodes represent the Basic Blocks and the arcs represent the relations between the Basic Blocks. A typical example of program and its CFG are shown in Fig. 1 (a) and (b) respectively.

The CFCVE approach consists of the following steps:

Algorithm A:

1) Extracting the Basic Block and generating the CFG of the program.

2) Assigning a signature s_i, which is generated by a specific algorithm, to node v_i, in which $s_i \neq s_j (if\ i \neq j,\ i, j = 1, 2, \cdots, N)$, where N is the total number of nodes in the program.

3) Inserting a Virtual vertex vv_{ij} into each Edge $e_{i,j}$ and inserting the signature updating instructions into each Virtual Vertex.

4) Inserting the appropriate instructions into the start and end of each Vertex in order to update the signature in run-time phase.

5) Inserting the checking instructions into the end of each Vertex (Virtual Vertex) in order to detect the control flow errors.

In this section the proposed scheme is explained in details. Subsection 1 explains the signature generation for the basic blocks and subsection 2 introduce the insertion of Virtual Vertexes into Edges and the insertion of updating instructions into a Vertex (Virtual Vertex) are introduced. Insertion of the checking instructions into each Vertex is described in subsection 3, and the control flow checking scenario is described in subsection 4.

3.1 Signature Generation

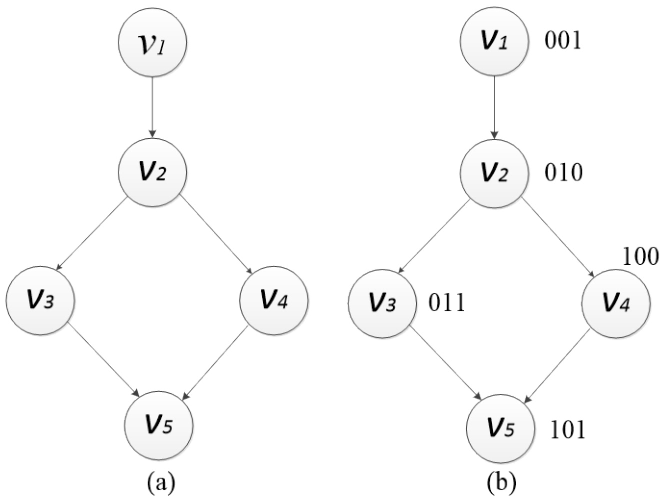


Fig. 2. The original CFG and the CFG with assigned signatures

Memory overhead is an important indicator in a control flow checking algorithm. In order to guarantee the minimum memory overhead, the length of the signature for each basic block should be as short as possible and try to use the least bit of the registers. In CFCVE we assign a binary positive number to each BB, and the length of a signature *Len* can be obtained by equation (1), where *N* is the number of total BBs.

$$Len = \lceil \log_2^N \rceil \tag{1}$$

Fig. 2 shows an example of generation signature for each node of CFG. **Fig. 2** (a) shows a sample program and **Fig. 2** (b) shows its corresponding program graph. A solid circle in the graph represents a basic block; the binary number on the right side of the solid circle is the signature of the corresponding node.

3.2 Visual edges creation and insertion of the updating instructions

CFCVE primarily focuses on internode CFEs. An illegal branch $br_{i,j}$ ($br_{i,j}$ is not included in E) from v_i to v_j indicates a CFE, i.e., each edge in E uniquely determines a legal branch $br_{i,j}$, therefore, the essence of a control flow checking technique is to determine whether the current branch $br_{i,j}$ is included in E. For this purpose, CFCVE transforms the structure of CFG by inserting a Virtual Vertex vv_{ij} into each Edge $e_{i,j}$. Note that Edges pointing from the vertexes to themselves should not have Virtual Vertexes inserted. **Fig. 3** shows an example of a control flow graph and the result of the transformation. The following is the complete description of Algorithm B, which creates a Visual Edge and inserts updating instructions into each Edge in a CFG.

```
Algorithm B:
For each Edge $e_{i,j}$, $i, j = 1,2,\ldots,N$.
  1) Generate a Virtual Vertex $vv_{i,j}$ whose successor vertex is $v_j$ and predecessor vertex is $v_i$ into $e_{i,j}$
  2) Insert Virtual Vertex $vv_{i,j}$ into $e_{i,j}$.
  3) Insert updating instructions $G = G \oplus s_i$ and $G = G \oplus s_j$ into Virtual Vertex $vv_{i,j}$.
```

Suppose that we mark the inserted Virtual Vertex $vv_{i,j}$ as vv_p and that s_i and s_j are the signatures of the source node v_i and the destination node v_j of branch $br_{i,j}$. After inserting the Virtual Vertex $vv_{i,j}$, the original Edge $e_{i,j}$ is divided into Edge e_{ip} and Edge $e_{p,j}$, i.e., the original branch $br_{i,j}$ is divided into branch $br_{i,p}$ and branch $br_{p,j}$. Since the Virtual Vertex vv_p contain no other instructions except an unconditional jump instruction (the destination node is v_j), once branch $br_{i,p}$ is taken, the control flow will transfer to Vertex v_j. Thus, the original Edge $e_{i,j}$ can be observed as a Virtual Edge in which a Virtual Vertex vv_p is embedded. An example of creating a virtual edge is showed in **Fig. 3**. The dotted circle in **Fig. 3** shows a Virtual Vertex. After inserted a Virtual Vertex, the original edge is transformed into a virtual edge. Virtual edges indicate that as execution of their origins completes, control flow continues from their terminals. In other words, they do not damage the control flow nature of original edges. For example for Virtual Edge $VE_{1,2}$ of **Fig. 3**, the control flow transfer from node v_1, through the Virtual Vertex $vv_{1,2}$, and reach to the node v_2. Obviously, the control flow of $e_{1,2}$ is not affected.

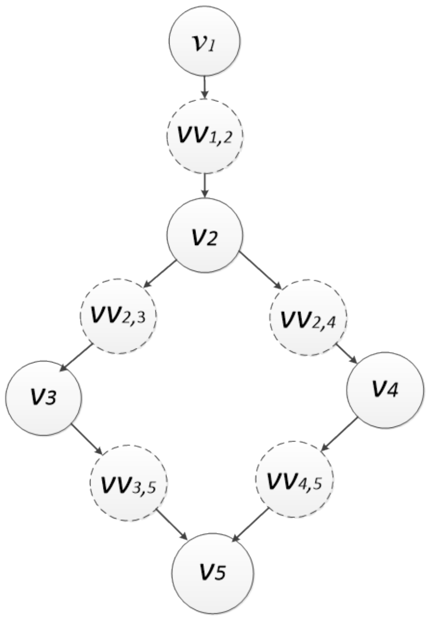


Fig. 3. An example of creating a virtual edge

$$\boxed{\begin{array}{l} G = G \oplus s_i \\ G = G \oplus s_j \\ \text{JUMP } v_j \end{array}}$$

Fig. 4. A Virtual Vertex with signature updating instructions

In order to update the run-time signature G associated with the current node of CFG, two signature updating instructions $G = G \oplus s_i$ and $G = G \oplus s_j$, are inserted into Virtual Vertex vv_p. Before the branch $br_{i,p}$ is taken, G is equal to G_i, which is the same as s_i. After the branch $br_{i,p}$ is taken, the control is transferred to vv_p, and the first signature updating instruction $G = G \oplus s_i$ is executed. Then, G is updated with a new run-time signature G_p, $G = G_p = G \oplus s_i = s_i \oplus s_j = 0$, and then the second signature updating instructions $G = G \oplus s_j$ is executed, after which $G = G \oplus s_j = 0 \oplus s_j = s_j$. Thus, when the control transfers to v_j, the run-time signature G is equal to the signature of node v_j. A Virtual Vertex with signature updating instructions is shown in **Fig. 4**.

3.3 Insertion of checking instructions into each vertex

To check the control flow, the checking instruction '$\mathbf{br\ (G \neq s_i)\ error}$' is inserted into the top of each BB. In other words, this checking instruction is executed prior to the execution of the original instructions in the BB; therefore, fault detection latency inside BB is avoided. As shown in **Fig. 5**, the basic block consists of the original instructions and with additional checking instruction located at its top.

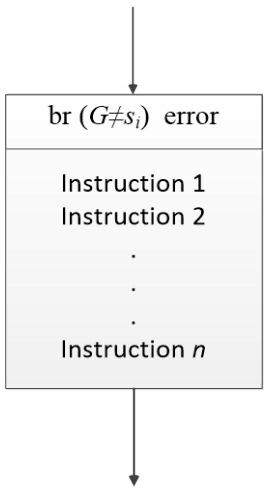


Fig. 5. A basic block with checking instruction

3.4 The control flow checking scenario

When control is transferred from one Virtual Vertex vv_k to its successor vertex v_i, the program execution reaches the start of node v_i first. Before execution of the signature checking instruction, the value of G is updated to G_i, and then the signature checking instruction is executed. If G_i is not equal to the signature s_i of vertex v_i, then a control flow error has occurred and control will be transferred to the error handling routine. On the other hand, if G_i is equal to s_i, it tells us there is no control flow error, then the rest instructions of vertex v_i will be executed. After that, the control is transferred to Virtual Vertex vv_p, which is the successor vertex of v_i. After the branch $br_{i,j}$ is taken, the first signature updating instruction $G = G \oplus s_i$ is executed, and then G is updated with a new run-time signature G_p, $G = G_p = G \oplus s_i = s_i \oplus s_i = 0$, the value of G is updated to 0. Then the second signature updating instructions $G = G \oplus s_j$ is executed where s_j is the successor vertex of v_j, $G = G \oplus s_j = 0 \oplus s_j = s_j$. Next, the unconditional jump instruction of Virtual Vertex vv_p is executed, and control is transferred to vertex v_j.Thus, when the control is transferred to v_j, the run-time signature G is equal to the signature of node v_j.

4. Enhancement of Methodology

4.1 The Entry/Exit bit

It has been shown that illegal branches violating the control flow entered at the beginning of each Basic Block can be detected by CFCVE. However, there are cases where legal branches entering somewhere inside a basic block cannot be detected. For example, in **Fig. 6**, node v_1 is the predecessor of node v_2. After executing the last instruction of node v_1, the branch $br_{1,2}$ will take; the control will transfer to node v_2. There is no problem when $br_{1,2}$ enters at the beginning of node v_2; however, intranode CFE occurs if $br_{1,2}$ enters at the middle of node v_2, and this intranode CFE cannot be detected by current CFCVE.

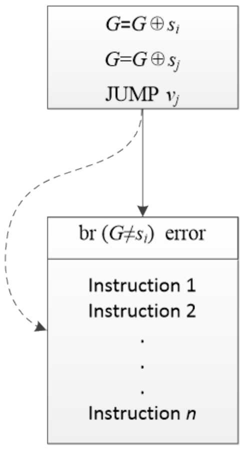


Fig. 6. The example of control flow transfers to the middle of node

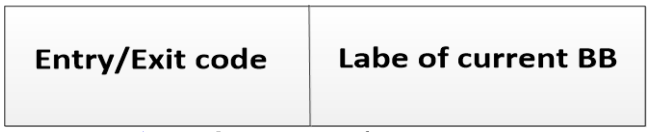


Fig. 7. The structure of a new signature

In order to enhance the Error detection capabilities of CFCVE, the Entry/Exit bit, which can detect the control flow errors that jump to or from the middle of a BB is introduced. The length of the Entry/Exit bit is 1 bit, and the value is set to 1/0 when the execution reaches to the entrance/exit of each BB. Thus, the new signature for each basic block consists of two sections; the Entry/Exit code and the original signature, i.e., Label of current BB (see the **Fig. 7**).

We call the new signature BS/ES when the value of the Entry/Exit bit is equal to 1/0. For each BB, a signature updating instruction $G = G \oplus BS_i$ is inserted behind the original checking instruction and then a signature updating instruction $G = G \oplus ES_i$ is inserted into the end before the last instruction of Vertex v_i . In addition, the original checking instruction ' $\mathbf{br}\,(G \neq s_i)\,\mathbf{error}$ ' is changed to ' $\mathbf{br}\,(G \neq BS_i)\,\mathbf{error}$ '. Moreover, the original signature updating instructions $G = G \oplus s_i$ and $G = G \oplus s_j$ are changed to $G = G \oplus ES_i$ and $G = G \oplus BS_j$ respectively.

4.2 Control flow checking scenario

When control is transferred from one Virtual Vertex v_k to its successor vertex v_i , the signature checking instruction is executed. If G_i is not equal to the signature BS_i of vertex v_i , then a control flow error has occurred and control will be transferred to the error handling routine. However, if G_i is equal to BS_i , there is no control flow error. Then the program execution reaches the start of node v_i , before execution of the first signature updating instruction, the value of G will be equal to BS_i in the absence of any errors. Then, the first signature updating instruction is executed, and the value of G is updated to 0; next, the remaining instructions of node v_i will be executed and after the second signature updating instruction is executed, the value of G will be set to ES_i . When control is transferred from vertex v_i to its successor Virtual Vertex v_p , the program execution reaches the start of v_p first. Before execution of the first signature updating instruction, the value of G will be equal to ES_i in the absence of any errors. Next, the first signature updating instruction is executed and the value of G is updated to 0, then the second signature updating instruction is executed, and the value of G will be set to BS_j .

5. Error detection capabilities

As mentioned in section 3.2, CFCVE primarily focuses on internode CFEs. An internode CFE falls into one of the following two cases:

- a branch $br_{i,j}$ to a basic block v_j not belonging to $suc(v_i)$
- a branch $br_{i,j}$ to somewhere inside a basic block v_j belonging to $suc(v_i)$

CFCVE is capable of detecting all internode CFEs.
Proof:

Type 1: a branch $br_{i,j}$ to a basic block v_j not belonging to $suc(v_i)$

Suppose that $br_{i,j}$ is an illegal branch, and $v_j \notin suc(v_i)$.

If v_i is a Virtual Vertex, then at node v_i G is equal to BS_k , where $v_k \in suc(v_i)$. Before the branch $br_{i,j}$ is taken, the new run-time signature is generated, $G = G \oplus BS_k = 0 \oplus BS_k = BS_k$. If v_j is a Virtual Vertex and $v_m \in suc(v_j)$, after the branch is taken, the two signature updating instructions of node v_i are executed. Then, the branch $br_{j,m}$ is taken, and the checking instruction ‘ $\text{br}(G \neq BS_m)\text{error}$ ’ of node v_m is executed. G is not equal to BS_m , because the signature of CFCVE is unique. Control is then transferred to the error handler, thus the error is detected. If v_j is not a Virtual Vertex, then after the branch $br_{i,j}$ is taken, the checking instruction ‘ $\text{br}(G \neq BS_j)\text{error}$ ’ is executed. Since $G = BS_k \neq BS_j$ control is transferred to the error handler, thus the mismatch is detected.

The situation of v_i is not a Virtual Vertex is similar with above.

Type 2: a branch to somewhere inside a basic block v_j belonging to $suc(v_i)$

Suppose that $br_{i,j}$ is an illegal branch and the branch is taken to the middle of the node v_j , i.e., skipping the signature checking instruction and the first signature updating instruction.

If v_i is a Virtual Vertex, then v_j is not a Virtual Vertex, and at node v_i G is equal to BS_j . After the branch is taken, the run-time signature is still equal to BS_j , and then the remaining instructions including the second signature updating instruction are executed, and the new run-time signature is generated, $G = G \oplus ES_j = BS_j \oplus ES_j$. After the instructions in node v_j are executed, $br_{j,k}$ is taken, where $v_k = suc(v_j)$, and v_k is a Virtual Vertex. The signature updating instructions in v_k update the value of $G = BS_j \oplus ES_j \oplus ES_j \quad \oplus BS_m$, where $v_m = suc(v_k)$ and v_m is not a Virtual Vertex. After $br_{k,m}$ is taken, the checking instruction ‘ $\text{br}(G \neq BS_m)\text{error}$ ’ is executed. Thus, the error is detected due to $BS_j \oplus ES_j \oplus ES_j \oplus BS_m = BS_j \oplus BS_m \neq BS_m$.

If v_i is not a Virtual Vertex, then v_j is a Virtual Vertex, and at node v_i G= is equal to ES_i . After the branch is taken, G will be updated to BS_k in the absence of control errors, where $v_k = suc(v_j)$ and v_k is not a Virtual Vertex, by the signature updating instructions of v_j . Suppose that $br_{i,j}$ is taken to the second signature updating instruction and skipping the first signature updating instruction. Then, $G = G \oplus BS_k = ES_i \oplus BS_k$ is generated, and the unconditional jump instruction is executed. The control is transferred to v_m , where $v_m = suc(v_k)$ and v_m is not a Virtual Vertex. At node v_m , the checking instruction ‘ $\text{br}(G \neq BS_m)\text{error}$ ’ is executed. Since $G = ES_i \oplus BS_k \neq BS_m$, the error is detected.

6. Experimental evaluation

In order to assess the effectiveness of the proposed approach, four benchmark programs are chosen for the experiment: Quick sort (QS), Bubble Sort (BS), Matrix Multiplication (MM), Fast Fourier Transformation (FFT). These target programs are chosen for a certain set of reasons. Firstly, they present certain varieties of control flow graph patterns. QS and BS are branch intensive programs. They have small size Basic Blocks and take a lot of branching among relatively simple calculations, resulting in more substantial overheads. MM and FFT are calculation intensive programs that perform substantial time-consuming multiplication and much less branching. Thus, the overhead of checking instructions is smaller compared to branch intensive programs. At the same time, these target programs use almost all instructions available, including the ones for arithmetic and logic calculations, and branching. Meanwhile, the target programs are merely the most common standardized algorithms that see a large number of applications, which makes them even more representative.

The proposed solution is experimentally evaluated on an ARM920T microprocessor machine running Linux kernel 2.6.32. The microprocessor has 3GB SDRAM and runs at a frequency of 400 MHz. Memory overhead, performance overhead, error detection latency and error detection coverage are imperative parameters for evaluating our approach. Thus, all of these parameters are measured and reported. First, the memory overhead and performance loss results of the presented scheme are evaluated, and then the average error detection latency of the presented scheme is analyzed and the last part allotted to error detection coverage. We considered four versions for each benchmark:

- the original code,
- a safe one, obtained by applying the CFCSS [15] technique to the original code,
- a safe one, obtained by applying the RSCFC [14] technique to the original code,
- a safe one, obtained by applying the CFCVE technique to the original code.

Each program (totally 16) is compiled and executed for 500 times. Memory overhead and performance overhead are compared between the hardened programs and the original ones , we determined the overheads recorded in Table 2.

As shown in **Table 2**, CFCVE incurs the least memory overhead (2.21%) compared with CFCSS (2.44%) and RSCFC (2.35%). In terms of performance overhead, the CFCVE (10.60%) is comparable to the CFCSS (10.55%), but less than the RSCFC (18.45). The memory overhead is mainly caused by signature overhead of these target programs. This memory overhead also affect on program performance. The extra execution time for the signature updating and signature checking instructions of these target programs is considered performance overhead.

Table 2. Memory overhead, performance overhead and error detection latency comparison

Program	Memory overhead (%)			Performance overhead (%)			Detection latency (cycle)		
	CFCSS	RSCFC	CFCVE	CFCSS	RSCFC	CFCVE	CFCSS	RSCFC	CFCVE
QS	2.42	2.93	2.24	14.50	25.0	11.0	4.70	5.72	5.20
BS	2.71	2.13	2.46	11.00	16.30	10.50	4.56	5.51	5.40
MM	2.17	1.36	1.94	6.50	14.20	5.51	5.63	7.16	6.51
FFT	2.47	2.96	2.25	10.20	18.30	15.42	5.37	7.37	7.12

In CFCSS, each node is assigned a unique signature and has two additional instructions. When the node is a branch-fan-in node a run-time adjusting signature and another two additional instructions are introduced to solve the aliasing of signatures. In CFCVE, only one unique signature is employed to identify each node, and three additional instructions are used to detect control flow errors. In order to solve the aliasing of signatures, a Virtual Vertex containing three instructions is inserted into each Edge. Thus, due to the introduction of the run-time adjusting signature, the memory overhead of CFCSS is larger than for CFCVE; the instructions overhead is considerable, even the number of instructions is relatively small. In RSCFC, each block takes one bit in the signatures of basic blocks. When the total number of basic blocks in a program is large, the signature of each block will have many bits, possibly exceeding the limitation of machine word possibly. In order to deal with this situation, basic blocks are grouped into multiple hierarchies. Thus, a hierarchy signature is introduced and hierarchy signature should be checked in each node. This process leads to a higher memory overhead than CFCVE. In addition, the local cumulative signature checking instructions increase the performance overhead greatly. Therefore, RSCFC incurs more performance overhead than CFCVE.

The error detection latency is the latency between fault occurrence and error detection. It may cause erroneous output and can directly affect rollback recovery. Therefore, the errors should be detected and addressed before erroneous output occurs. Detection latency can be quantified in processor cycles, and is defined here as the number of processor cycles to run between the terminal of the illegal branching and the line detecting the error here. The detection latency result in **Table 2** is calculated according to the equation 2 and 3 of [17]. **Table 2** shows that the detection latency of CFCVE is comparable to RSCFC, but much higher than for CFCSS. The reason is that the number of additional instructions in CFCSS is 2-4, 6 in CFCVE and 7 in RSCFC.

We adopted a method in which the faults are injected into the program through modifying the assembly codes of the source file [23, 24]. For each program (totally 16), the source file is compiled and assembly code is generated firstly. Secondly, one of the branch deletion, branch creation or branch operand changes was randomly applied to the assembly codes. Finally, the resulting assembly code is compiled and executed. Each kind of fault is injected 2000 times into the original program.

The results of fault injection and fault detection we gathered during fault injection experiments are reported in **Tables 3** and **4**. Transient faults injected into the unhardened programs are categorized according to their effects (**Table 3**) and then compared with the results of injection into the 3 safe versions (CFCSS, RSCFC and CFCVE), as reported in **Table 4**. Fault detection effects are classified as follows:

- Wrong Result (WR): the fault modifies the results of the program without being detected.
- OS detection (OS): the fault is detected by the operating system or the hardware.
- Software Detection (SD): the fault is detected by the software detection mechanisms employed.

The undetected incorrect outputs are gotten by WR and error detection coverage is gotten by SD. **Fig. 8** was generated based on the WR in **Tables 3** and **4** and illustrates the fraction of faults that are not detected for the original programs and the hardened programs with CFCSS, RSCFC and CFCVE under each fault types.

Table 3. Experiment results of fault detection effects in original programs.

Original Program	**Del (2000#) (%)**		**Change (2000#) (%)**		**Insert (2000#) (%)**	
	OS	**WR**	**OS**	**WR**	**OS**	**WR**
QS	31.2	36.7	71.2	25.4	52.4	28.7
BS	35.1	48.7	67.9	26.7	57.9	16.7
MM	32.5	40.3	65.1	25.3	58.6	25.3
FFT	36.6	40.9	64.5	30.9	64.5	26.5

Table 4. Experiment results of fault detection effects in programs with CFCSS, RSCFC and CFCVE.

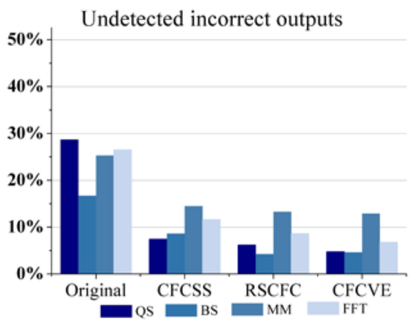
Programs	**Del (2000#) (%)**			**Change (2000#) (%)**			**Insert (2000#) (%)**		
	OS	**WR**	**SD**	**OS**	**WR**	**SD**	**OS**	**WR**	**SD**
QS-CFCSS	22.3	17.4	38.2	21.8	10.5	58.2	25.1	7.46	56.2
BS-CFCSS	14.2	6.73	38.6	12.7	16.9	68.1	15.2	8.64	64.3
MM-CFCSS	19.4	12.7	37.5	16.1	5.30	65.4	28.7	14.5	50.4
FFT-CFCSS	13.4	20.3	53.2	11.5	13.4	73.7	21.4	11.6	60.1
QS-RSCFC	19.3	15.1	43.5	15.6	7.82	65.7	17.1	6.20	65.4
BS-RSCFC	4.63	5.34	60.2	8.64	13.2	76.3	12.5	4.24	71.4
MM-RSCFC	15.2	7.60	47.8	13.1	4.83	69.4	23.5	13.3	54.6
FFT-RSCFC	9.82	15.4	62.3	8.16	11.6	76.3	13.7	8.64	65.5
QS-CFCVE	21.4	15.5	49.0	14.4	8.34	64.6	16.5	4.75	67.7
BS-CFCVE	6.40	4.20	55.5	8.25	14.5	75.4	12.8	4.64	72.2
MM-CFCVE	13.6	8.92	48.1	15.5	5.35	67.7	22.5	12.9	55.4
FFT-CFCVE	13.1	14.9	59.5	7.12	10.8	78.6	15.8	6.78	61.3

As it can be seen in Fig. 8, the average undetected incorrect outputs for CFCSS, RSCFC and CFCVE are 12.11%, 9.44% and 9.29% respectively. The figures of average error detection coverage are 55.32%, 63.20%, and 62.91%, respectively. As mentioned before, memory overhead, performance overhead, error detection latency and error detection coverage are imperative parameters for evaluating our approach. In order to balance these parameters a new parameter, called Evaluation Factor (EF), is introduced in Literature [25]. However, the error detection latency is not taken into account. We redefined the Evaluation Factor; the new definition of Evaluation Factor is showed in equation 2.

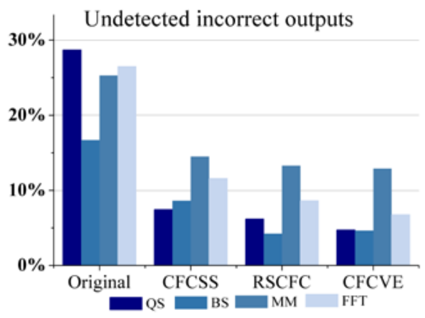
$$\mathrm{EF} = \frac{\text{Error detection coverage}}{\text{Memory Overhead} \times \text{Performance Overhead} \times \text{Error detection latency}} \tag{2}$$

The averages of the Evaluation Factors are 0.42, 0.23 and 0.44 respectively. Thus, CFCVE is comparable in fault coverage the best of the previously proposed techniques. Meanwhile it has a higher evaluation factor because of the lower memory overhead and performance overhead compared with other methods.

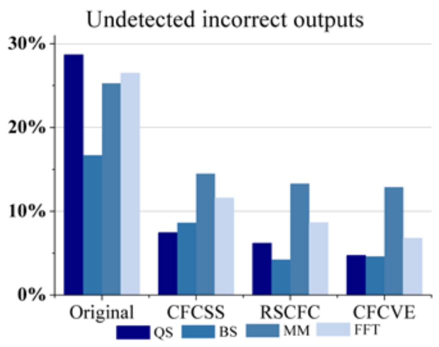
The following reasons can account for the excellent performance of CFCVE. First, in CFCSS, if multiple nodes are sharing multiple branch-fan-in nodes as their destination nodes, aliasing may occur between legal and illegal branches, and cause an undetectable control flow error. CFCVE can solve the aliasing by updating signature at the directed edges instead of at the nodes of control flow graph. Thus, the error detection coverage is higher than CFCSS. Second, in RSCFC, each block takes one bit in the signatures of basic blocks. The length of signature and the total number of basic blocks are in the direct ratio. This will incur substantial memory overhead when the quantity is large. Therefore, CFCVE incurs less memory overhead than RSCFC. Third, the memory overhead and error detection latency are lower due to the moderate additional instructions of CFCVE.



(a) branch deletion faults



(b) branch change faults



(c) branch creation faults

Fig. 8. The comparison of undetected incorrect outputs percentage under three fault types

7. Conclusions and Future Research

This article proposes a software-based control flow checking technique CFCVE. This method assigns a unique signature occupying the least bit to each basic block and updates the signatures at the virtual edges of the control flow graph. A Virtual Edge is introduced to resolve the signature aliasing. Fault and error injection experiments were conducted to assess the effectiveness of the proposed approach. The experiments showed that the CFCVE technique incurs the least memory overhead and performance overhead. The error detection latency is moderate. Error injection experiments on benchmarks showed that CFCVE can detect all the single inter-block control flow errors and has higher method efficiency [26] than previous techniques. However, Virtual Vertex introduces addition branches into the code, which may be affected by faults themselves.

Research is being conducted to develop excellent methods to protect checking instruction and signature updating instructions themselves in order to improve error coverage without increasing performance overhead. The trade-off between error detection latency and performance overhead is also a new research topic for our research group.

8. Acknowledgment

This research was supported by the National Natural Science Foundation of China under grant No. 61370134, the National High Technology Research and Development Program of China (863 Program) under grant No. 2013AA013901.

References

[1] Chen K, Liu H Y, Chen X S, "Detecting LDoS Attacks based on Abnormal Network Traffic [J]," *Ksii Transactions on Internet & Information Systems*, 6(7):1831-1853, 2012. Article (CrossRef Link)

[2] Ktas E, Athanasopoulos E, Bos H, et al., "Out of Control: Overcoming Control-Flow Integrity[C]," *IEEE Symposium on Security and Privacy. IEEE Computer Society*, 575-589, 2014. Article (CrossRef Link)

[3] Davi L, Sadeghi A R, Winandy M., "Dynamic integrity measurement and attestation: towards defense against return-oriented programming attacks.[C]," *ACM Workshop on Scalable Trusted Computing, Stc 2009*, Chicago, Illinois, Usa, 49-54, November. 2009. Article (CrossRef Link)

[4] Nagarajan A, Varadharajan V., "Dynamic trust enhanced security model for trusted platform based services [J]," *Future Generation Computer Systems*, 27(5):564-573, 2011. Article (CrossRef Link)

[5] Winter J, Dietrich K., "A hijacker's guide to communication interfaces of the trusted platform module [J]," *Computers & Mathematics with Applications*, 65(5):748-761, 2013. Article (CrossRef Link)

[6] Kanuparthi A K, Zahran M, Karri R., "Architecture Support for Dynamic Integrity Checking[J]," *IEEE Transactions on Information Forensics & Security*, 7(7):321-332, 2012. Article (CrossRef Link)

[7] Muthukumaran D, Schiffman J, Hassan M, et al., "Protecting the integrity of trusted applications in mobile phone systems [J]," *Security & Communication Networks*, 4(6):633-650, 2011. Article (CrossRef Link)

[8] Shacham H., "The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86)[C]," in *Proc. of ACM Conference on Computer and Communications Security, CCS 2007*, Alexandria, Virginia, Usa, 552-561, October. 2007. Article (CrossRef Link)

[9] Bhattacharya K, Ranganathan N., "RADJAM: A Novel Approach for Reduction of Soft Errors in Logic Circuits.[C]," in *Proc. of International Conference on Vlsi Design*, 453-458, 2009. Article (CrossRef Link)

[10] Saxena N R, Mccluskey E J., "Control-Flow Checking Using Watchdog Assists and Extended-Precision Checksums[J]," *Computers IEEE Transactions on*, 39(4):554-559, 1990. Article (CrossRef Link)

[11] Rajabzadeh A, Miremadi S G., "A Hardware Approach to Concurrent Error Detection Capability Enhancement in COTS Processors[C]," *Pacific Rim International Symposium on Dependable Computing, 2005. Proceedings. IEEE*, 83-90, 2005. Article (CrossRef Link)

[12] Jafari-Nodoushan M, Miremadi S G, and Ejlali A., "Control-Flow Checking Using Branch Instructions.[C]," *Ieee/ipip International Conference on Embedded and Ubiquitous Computing*, 66-72, 2008. Article (CrossRef Link)

[13] Alkhalifa, Z, Nair, V.S.S, Krishnamurthy, N, et al., "Design and evaluation of system-level checks for on-line control flow error detection[J]," *IEEE Transactions on Parallel & Distributed Systems*, 10(6):627-641, 1999. Article (CrossRef Link)
Li A, Hong B., "On-line control flow error detection using relationship signatures among basic blocks[J]," *Computers & Electrical Engineering*, 36(1):132-141, 2010. Article (CrossRef Link)

[14] Oh N, Shirvani P P, Mccluskey E J., "Control-flow checking by software signatures[J]," *IEEE Transactions on Reliability*, 51(1):111-122, 2002. Article (CrossRef Link)

[15] Jian-Li L I, Tan Q P, Tan L F, et al., "A Control Flow Checking Method based on Abstract Basic Block and Formatted Signature [J]," *Chinese Journal of Computers*, 2014. Article (CrossRef Link)

[16] Mu Y, Hao W, Zheng Y, et al., "Graph-tree-based software control flow checking for COTS processors on pico-satellites[J]," *Chinese Journal of Aeronautics*, 26(2):413-422, 2013. Article (CrossRef Link)

[17] Chielle E, Rodrigues G S, Kastensmidt F L, et al., “S-SETA: Selective Software-Only Error-Detection Technique Using Assertions [J],” *IEEE Transactions on Nuclear Science*, 62(6):3088-3095, 2015. Article (CrossRef Link)

[18] Martinez-Alvarez A, Restrepo-Calle F, Cuenca-Asensi S, et al., “A Hardware-Software Approach for On-line Soft Error Mitigation in Interrupt-Driven Applications[J],” *IEEE Transactions on Dependable & Secure Computing*, 502-508, 2016. Article (CrossRef Link)

[19] Watson M, Shirazi N, Marnerides A, et al., “Malware Detection in Cloud Computing Infrastructures[J],” *IEEE Transactions on Dependable & Secure Computing*, 13(2):192-205, 2016. Article (CrossRef Link)

[20] Venkatasubramanian R, Hayes J P, and Murray B T., “Low-cost on-line fault detection using control flow assertions[C],” in *Proc. of On-Line Testing Symposium*, Iolts. IEEE, 137-143, 2003. Article (CrossRef Link)

[21] Goloubeva O, Rebaudengo M, Reorda M S, et al., “Soft-Error Detection Using Control Flow Assertions[J],” *Nonlinear Dynamics*, 77(4):581-588, 2003. Article (CrossRef Link)

[22] Y. Sedaghat, S. G. Miremadi, M. Fazeli, “A Software-Based Error Detection Technique Using Encoded Signatures [J],” 389-400, 2006. Article (CrossRef Link)

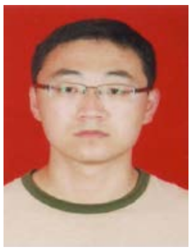
[23] Krishnamurthy N, Jhaveri V, and Abraham J., “A Design Methodology for Software Fault Injection in Embedded Systems [J],” 1998.

[24] Asghari S A, Taheri H, Pedram H, et al., “Software-Based Control Flow Checking Against Transient Faults in Industrial Environments [J],” *IEEE Transactions on Industrial Informatics*, 99(1):481-490, 2013. Article (CrossRef Link)

[25] Vemu R, and Abraham J., “CEDA: Control-Flow Error Detection Using Assertions [J],” *IEEE Transactions on Computers*, 60(9):1233-1245, 2011. Article (CrossRef Link)

LiPing Liu received the B.S. degree and M.S. degree in the Dept. of Computer Science from North University of China, China, in 2008 and 2011, respectively. Currently, he is studying for his PhD Degree at Computer Science in Beijing Institute of Technology. His current research interests include Secure Wireless Sensor Networks, Pattern Recognition, and Trusted Computing.

LinLin Ci received the B.S. degree in the Dept. of Computer Science from Beijing Institute of Technology, China, in 1976; he received the M.S. degree in the Dept. of Computer Science from Northwestern Polytechnical University, China, in 1985. Currently, he is professor and doctoral supervisor in computer application. His research areas include Secure Wireless Sensor Networks, Pattern Recognition, and Trusted Computing.



Wei Liu received the B.S. degree and M.S. degree from North University of China from Dept. of Computer Science, China, in 2008 and 2011, respectively. Currently, he is studying for his PhD Degree at Computer Science in Beijing Institute of Technology. His current research interests include Secure Wireless Sensor Networks, Pattern Recognition, and Trusted Computing.

Hui Yang received the B.S. degree and M.S. degree from North University of China from Dept. of Computer Science, China, in 2008 and 2011, respectively. Currently, he is studying for his PhD Degree at Computer Science in Beijing Institute of Technology. His current research interests include Secure Wireless Sensor Networks, Pattern Recognition, and Trusted Computing.