Control-Flow Checking by Software Signatures

Nahmsuk Oh, *Member, IEEE*, Philip P. Shirvani, *Member, IEEE*, and Edward J. McCluskey, *Life Fellow, IEEE*

***Abstract*—This paper presents a new signature monitoring technique, CFCSS (Control Flow Checking by Software Signatures); CFCSS is a pure software method that checks the control flow of a program using assigned signatures. An algorithm assigns a unique signature to each node in the program graph and adds instructions for error detection. Signatures are embedded in the program during compilation time using the constant field of the instructions and compared with run-time signatures when the program is executed. Another algorithm reduces the code size and execution time overhead caused by checking instructions in CFCSS.**

A "branching fault injection experiment" was performed with benchmark programs. Without CFCSS, an average of 33.7% of the injected branching faults produced undetected incorrect outputs; however, with CFCSS, only 3.1% of branching faults produced undetected incorrect outputs. Thus it is possible to increase error detection coverage for control flow errors by an order of magnitude using CFCSS.

The distinctive advantage of CFCSS over previous signature monitoring techniques is that CFCSS is a pure software method, i.e., it needs no dedicated hardware such as a watchdog processor for control flow checking. A watchdog task in multitasking environment also needs no extra hardware, but the advantage of CFCSS over a watchdog task is that CFCSS can be used even when the operating system does not support multitasking.

***Index Terms*—Assigned signatures, control flow checking, fault injection experiments, signature monitoring, software error detection.**

ACRONYMS[1]

ARGOS	Advanced Research and Global Observation Satellite
CF	control flow
CFCSS	CF checking by software signatures
CPU	central processing unit
GSR	global signature-registers
SIC	structural integrity checking.

NOTATION

V	$\{v_i: i = 1, 2, \ldots, n\}$: set of vertices denoting basic blocks
E	set of edges denoting possible CF between basic blocks
P	program graph $\{V, E\}$
s_i	signature of v_i
d_i	signature difference in v_i
G	run-time signature
G_i	value of G in v_i
D	run-time adjusting signature
$\text{br}_{i,j}$	a branch from v_i to v_j
$\text{suc}(v_i)$	set of successors of v_i
$\text{pred}(v_i)$	set of predecessors of v_i.

Manuscript received November 1, 1999; revised July 10, 2000. This work was supported in part by the Ballistic Missile Defense Organization, Innovative Science and Technology (BMDO/IST) Directorate and administered through the U.S. Department of the Navy, Office of Naval Research under Grants N00014-92-J-1782 and N00014-95-1-1047. Responsible Editor: R. Karri.

The authors are with the Center for Reliable Computing; Department of Electrical Engineering and Computer Science, Stanford University, Stanford, CA 94305 USA (e-mail: {NSoh;Shirvani; EJM}@crc.stanford.edu).

Publisher Item Identifier S 0018-9529(02)02954-8.

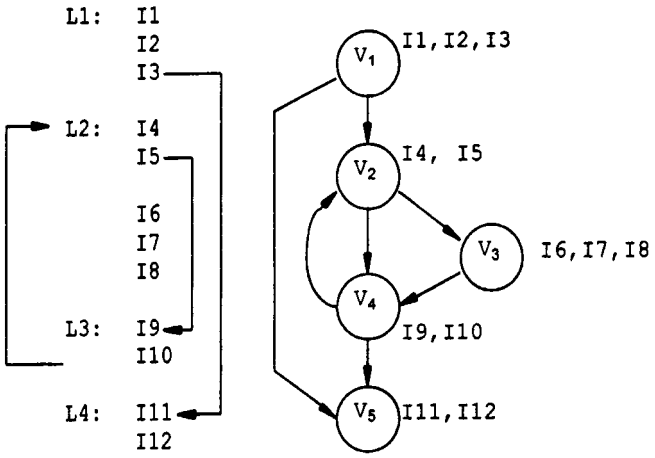
[1]The singular and plural of an acronym are always spelled the same.

TERMINOLOGY

- Basic Block: A maximal set of ordered instructions in which its execution begins from the first instruction and terminates at the last instruction. There is no branching instruction in a basic block except possibly for the last one. A basic block terminates at either an instruction branching to another basic block or an instruction receiving transfer of CF from two or more places in the program [3].
- Program Graph: From the definitions of V and E, a program can be represented by a program-graph, P. The $\text{br}_{i,j}$ are not necessarily explicit branch instructions; they also represent fall-through execution paths, jumps, subroutine calls, and returns. Fig. 1 is an example.
- Illegal: v_j is in the $\text{suc}(v_i)$ if and only if $\text{br}_{i,j}$ is included in E. Similarly, v_i is in $\text{pred}(v_j)$ if and only if $\text{br}_{i,j}$ is included in E. If a program is represented by its $P = \{V, E\}$, then $\text{br}_{i,j}$ (during the execution of P) is illegal if $\text{br}_{i,j}$ is not included in E [3]. This illegal branch indicates a CF error, which can be caused by transient or permanent faults in hardware such as the program counter, address circuits, or memory system [1].
- Branch-Fan-in Node, Branch Insertion, Branch Deletion: If a node receives more than 2 transfers of CF it is a branch-fan-in node, i.e., the number of nodes in $\text{pred}(v) > 1$. A branch-insertion occurs when one of the instructions in the node is changed to a branch-instruction as the result of an error. A branch-deletion occurs when an error causes the branch-instruction of a node to change to a nonbranch instruction. As a result, the node without the branch-instruction merges with the node that is adjacent to it in the memory address space.
- Xor-Difference of a and b: the result of performing the bitwise XOR operation of a and b, i.e., xor-difference $= a \oplus b$, where a and b are binary numbers.

I. INTRODUCTION

TRANSIENT or permanent faults introduced in a computer system during runtime can cause an incorrect sequence of instruction–execution in the program, and can cause CF errors. If the system does not perform some run-time checking, the erroneous output might not be detected and serious damage could



$E = \{\mathrm{br}_{12}, \mathrm{br}_{13}, \mathrm{br}_{24}, \mathrm{br}_{34}, \mathrm{br}_{42}, \mathrm{br}_{45}, \mathrm{br}_{15}\}$
$V = \{V_1, V_2, V_3, V_4, V_5\}$

Fig. 1. Sequence of instructions and its graph.

result. Therefore, it is important to monitor the program to detect any abnormality in the CF or other error, and to take appropriate actions to avoid any incorrect output.

This paper presents CFCSS, a new assigned signature monitoring technique which checks the inter-block CF of a program using instructions, but without using any special hardware. The program is divided into basic blocks. A program graph is constructed based on the CF of the program. A node in the program graph represents a basic block. All nodes in the program graph are assigned different arbitrary numbers (signatures), which are embedded into the program during preprocessing or compile time. During program execution, a run-time signature, G is stored in one of the general purpose registers, GSR, and compared with the stored signature of the node whenever control is transferred to a new node. For multiple branching, a runtime adjusting signature D is combined with G. The complete algorithm and an example program are in Section III.

CFCSS differs from other signature monitoring techniques in using no watchdog processor or extra hardware to achieve interblock CF checking. CFCSS uses an assigned signature technique similar to SIC [1], but does not need to send check-labels to a watchdog processor because it checks the signatures using instructions. Block Signature Self-Checking [2] is also an assigned signature technique that uses a subroutine to replace the watchdog processor. However, its drawback is that the code depends on the location of the code because the signature consists of an absolute address. The CF checking scheme in [3] is a pure software method but it constructs a database containing information about concurrent CF checking, thus it might require appreciable memory overhead. The watchdog task in [4] also needs no extra hardware, but the operating system must support a multitasking environment. On the other hand, a watchdog processor or similar monitoring hardware is required in the derived signature techniques as in

- Path Signature Analysis [5];
- Signatured Instruction Streams (SIS) [6];
- Asynchronous SIS [7];
- Continuous Signature Monitoring (CSM) [8], [9];
- extended-precision checksum method [10];
- On-line Signature Learning and Checking (OSLC) [11];
- Implicit Signature Checking (ISC) [12].

They check the intra-block and inter-block CF by observing the signatures derived from the instruction bit patterns or addresses of a basic block while CFCSS checks the inter-block CF by monitoring assigned signatures in the GSR. Although they have similar function, the GSR differs from the reserved-register in [13]; the reserved-register stores the program or procedure name whereas the GSR stores the signature of the current node. The GSR is not a special or additional register in the CPU. It is one of the general purpose registers of the CPU picked by the compiler or assembler to serve as the GSR. In addition, CFCSS differs from VASC [14] in the way it embeds and checks for the signatures.

The motivation of CFCSS came from a real space experiment, the Stanford ARGOS (Advanced Research and Global Observation Satellite) project [15]. The USA (Unconventional Stellar Aspect) experiment [16] is one of the experiments that the ARGOS satellite carries, and includes a computing test-bed that has two processor boards. One of the two boards was built using only commercial-off-the-shelf components and has no hardware for error detection; only pure software techniques can be applied to detect errors. In this case, the software technique CFCSS can be used to increase the system reliability by enhancing CF error detection capability.

II. Previous Work

Signature-monitoring and pure software methods have been proposed to check the CF of a computer system. Signature-monitoring is a method in which a signature associated with a block of instructions (one or more nodes in the CF graph) is calculated and saved somewhere during compile time; then, the same signature is generated during run time and compared with the saved one. Signatures are assigned arbitrarily (assigned signatures) or derived from the binary code or the address of the instructions (derived signatures). Structural Integrity Checking (SIC) [1] uses an assigned signature method while Path Signature Analysis (PSA) [5], Signatured Instruction Streams (SIS) [6], Asynchronous SIS (ASIS) [7], Continuous Signature Monitoring (CSM) [8], [9], extended-precision checksum method [10], and On-line Signature Learning and Checking (OSLC) [11] all use derived signatures.

In many signature monitoring techniques, dedicated hardware is used to calculate the run-time signatures and to compare them with the saved signatures. A watchdog processor is proposed for this purpose in [17], [18]. On the other hand, when the hardware is fixed and cannot be changed, a software method that does not need any extra hardware is necessary. Examples of software methods are:

- assertions [4], [19];
- watchdog task [4];
- Block Signature Self-Checking (BSSC) [11];
- Error Capturing Instructions (ECI) [11];
- timers to check the behavior of the program [20];
- Available Resource-driven Control-flow monitoring (ARC) [21];
- temporal redundancy methods [22].

An assigned signature monitoring technique, Versatile Assign Signature Checking (VASC), has been proposed and evaluated in [14], in which the comparison of the ratio between CF errors and data errors in RISC and CISC processors are reported.

III. Description of CFCSS

A. The Run-Time Signature G

CFCSS checks the CF of the program by using GSR, a dedicated register 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 v_i in the program flow graph) is identified and assigned a unique s_i when the program is compiled. Under normal execution of the program (no errors), G_i should equal s_i. If G contains a number different from the signature associated with the current node, then an error has occurred in the program.

When control is transferred from one basic block to another, a new G is generated by a signature function f at the destination node of the branch. A signature function f is a function that updates G for the current node by using two values:

- the signature of the previous node (source node of the branch); and
- the signature of the current node (destination node of the branch).

These two values are used because the source and destination nodes of the branch uniquely determine each branch in E.

Let the signature function

$$f \equiv f(G,\, d_i) = G \oplus d_i.$$

The $d_d \equiv s_s \oplus s_d$ is calculated in advance at compile-time and stored in v_d. Before the br_{sd} is taken, G contains G_s. After the branch is taken, G is updated with a new value, $G_d = f(G_s,\, d_d)$, based on the previous G_s and d_d. If $G_d = s_d of v_d$, then there is no CF error. On the other hand, if $G_d \neq s_d$, then a CF error has occurred.

The XOR operation was chosen as the signature function because the XOR operation is better than other ALU operations for the purpose of checking or generating signatures. Because AND, OR, XOR operations use fewer gates in the ALU than addition and multiplication, they have less chance of having an error in the ALU than addition and multiplication. Now, check the correct CF in the original program and minimize the probability of error in the signature function. The fewer gates the signature function uses, the lower the probability of an error in calculation of the signature functions. With AND and OR operations, given one input operand (d_d) and the output result (s_d), the other input operand (s_s) cannot be uniquely determined. Hence, signature aliasing can occur, resulting in undetected errors. XOR does not have this aliasing problem. Therefore, the best candidate for the signature function is the XOR operation.

For example, in Fig. 2, $f \equiv f(G,\, d_i) = G \oplus d_i$; the nodes are assigned unique numbers as their signatures. Before a branch is taken, $G = G_1$ that is the same as s_1, the signature of the source node of the branch. After the branch is taken, G is updated with a new run-time signature G_2:

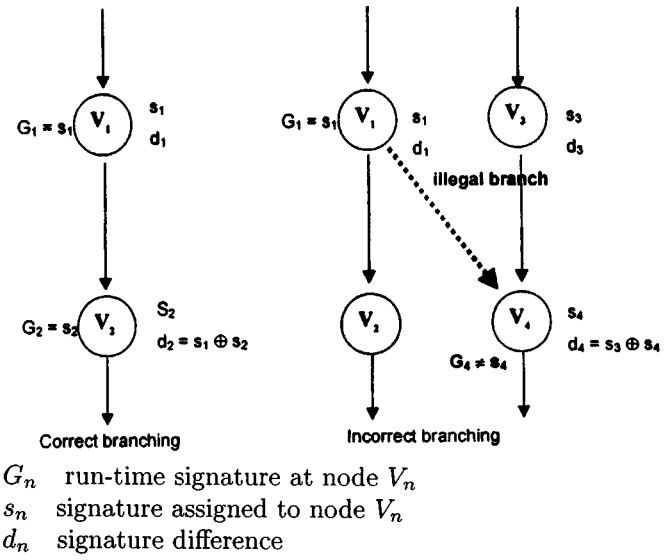


Fig. 2. Detection of an illegal branch.

$$G = G_2 = f(G_1,\, d_2) = G_1 \oplus d_2.$$

Because $d_2 = s_1 \oplus s_2$ and G_1 was s_1, the new run-time signature is

$$G_2 = f(G_1,\, d_2) = G_1 \oplus d_2 = s_1 \oplus (s_1 \oplus s_2) = s_2$$

i.e., the updated G_2 is the same as the s_2 of the current v_2; therefore, no error has occurred. On the other hand, suppose that an illegal branch from v_1 to v_4 is taken; i.e., the control should have moved from v_1 to node v_2, but an error causes an illegal branch from v_1 to v_4. Before the illegal branch is taken, $G_1 = s_1$, as before. However, after the branch is taken, at v_4, the new updated G_4 is different from the s_4 of the new v_4 because

$$\begin{aligned} G_4 &= f(G_1,\, d_4) = G_1 \oplus d_4 = G_1 \oplus (s_3 \oplus s_4) \\ &= s_1 \oplus (s_3 \oplus s_4) \neq s_4 \end{aligned}$$

i.e., the CF error can be detected by observing that the run-time signature is different from the new-node signature.

Before showing the exact algorithm, the outline of the algorithm for adding CFCSS to a program is described.

All basic blocks (nodes) in the program are identified and numbered. Each basic block is assigned a unique signature. The signature difference (XOR-difference between the source and destination nodes) of all branches is also calculated and stored in the destination nodes of all the branches. Whenever the control enters a new node, the run-time signature is updated to a G_{new} by the f that uses the previous run-time G_{prev} and the d_{new} as the arguments. If the G_{new} is the same as the signature of the new node, the instructions in the node are executed; otherwise, it means a CF error has occurred and control is transferred to an error handling routine.

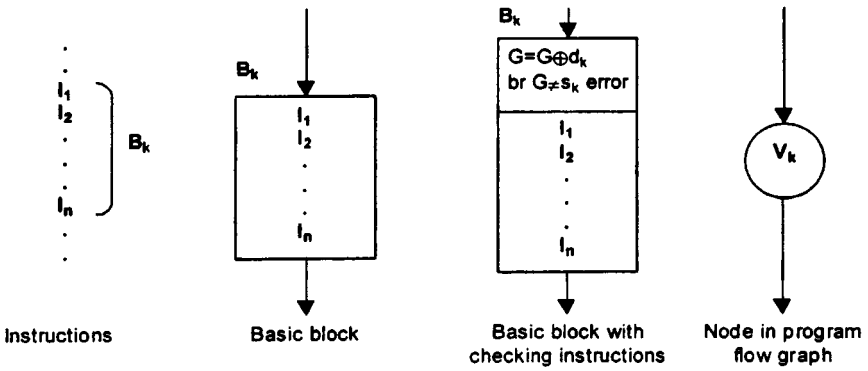


Fig. 3. Basic block with checking instructions.

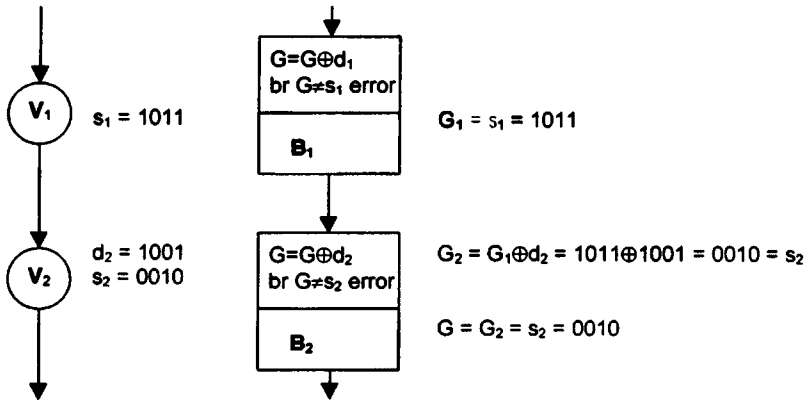


Fig. 4. Checking instructions in a correct control-flow.

Algorithm Details

To check the CF, checking instructions are located at the top of each basic block, i.e., checking instructions are executed prior to the execution of the original instructions in the basic block. In Fig. 3, the basic block B_k consists of instructions $I_1, I_2, \ldots, I_n$ and additional checking instructions located at its beginning. The checking instructions consist of 2 parts:

- the signature function that generates the run-time signature $(G = G \oplus d_k)$,
- the branch instruction, "br $(G \neq s_k)$ error," that compares the run-time signature with the signature of basic block B_k.

In this way, a v_k represents a B_k with its checking instructions.

Fig. 4 shows how the checking-instructions detect errors. The control is going to be transferred from v_1 to v_2. $G = G_1 = s_1 = 1011$, the signature of the current v_1. After the $\mathrm{br}_{1,2}$ is taken, the f generates the new run-time

$$G = G_2 = G_1 \oplus d_2 = 1011 \oplus 1001 = 0010,$$

and G is compared with the s_2 by the "br $(G \neq s_2)$ error" instruction. The conditional branch instruction "br $(G \neq s_2)$ error" branches to the error handler if G and s_2 are different.

Fig. 5, in contrast to Fig. 4, shows the case where an illegal branch is taken and how it is detected by the checking instructions. Before an illegal $\mathrm{br}_{1,4}$ is taken, G has the value s_1. However, at v_4, the new run-time signature $G = G_4$ is different from s_4 because $G = 0101$ and $s_4 = 0110$

$$(G_4 = G_1 \oplus d_4 = 1011 \oplus 1110 = 0101 \neq s_4 = 0110).$$

This mismatch causes the following instruction "br $(G \neq s_4)$ error" to transfer the control to the error handling routine.

Fig. 6, in contrast to Fig. 5, shows the case in which an error occurring in the branch instruction, e.g., a bit flip in the destination field, causes an unpredictable jump to any place in the entire program, *viz*, any basic block in the program and any place in that basic block. $G = s_1$ at node v_1. The illegal branch from v_1 to v_4 is taken, and the control is transferred to one of the instructions in the B_4, not the checking instructions. Because a new run-time signature is not generated at v_4, G still equals the previous value G_1. G is not updated to s_4 although control is transferred to v_4. After the instructions in v_4 are executed, the branch from v_4 to v_5 is taken. At v_5, G is updated to a new value G_5, but it is not equal to the signature of v_5 because the previous G before the $\mathrm{br}_{4,5}$ was $G_1(=s_1)$, not the correct value $G_4(=s_4)$. Thus an illegal branch to any instruction in the node is also caught. The detailed calculation of G is shown in Fig. 6.

Fig. 7 shows the case where an illegal branch $\mathrm{br}_{1,4}$ lands at instruction #2 of the node, "br $(G \neq s)$ error." In a similar way, because the new run-time signature is not generated at v_4, then G is still equal to the previous G_1 that is not equal to s_4. Therefore, "br $(G \neq s)$ error" catches this mismatch, and thus the error is detected.

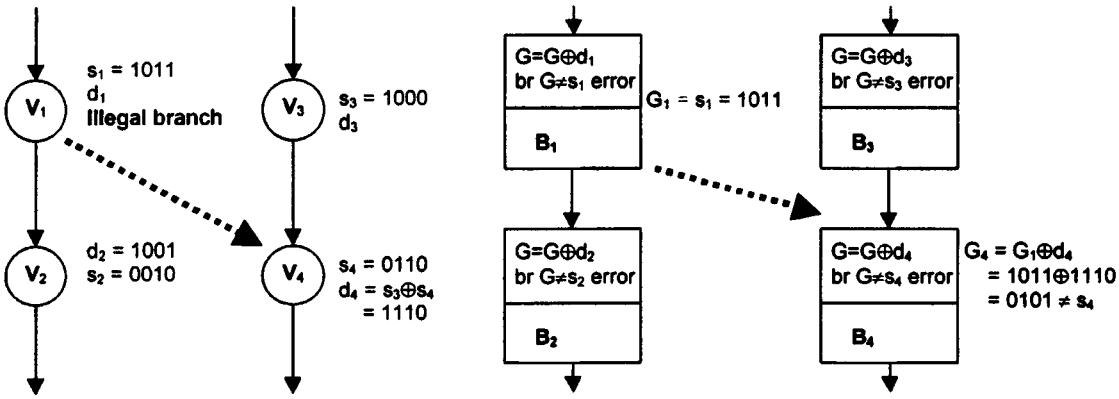


Fig. 5. Detection of an illegal branch.

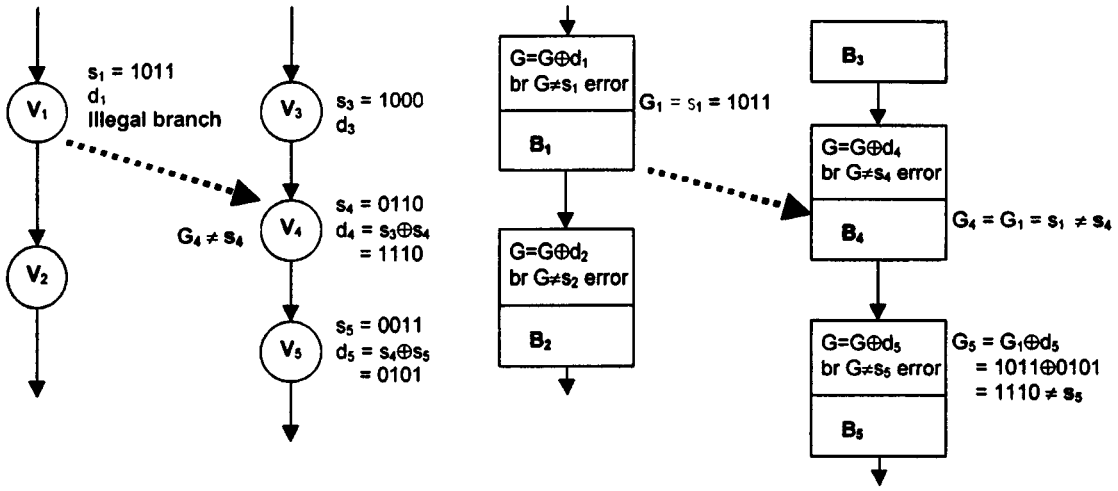


Fig. 6. Detection of a branch which is illegally jumping to the middle of a basic block.

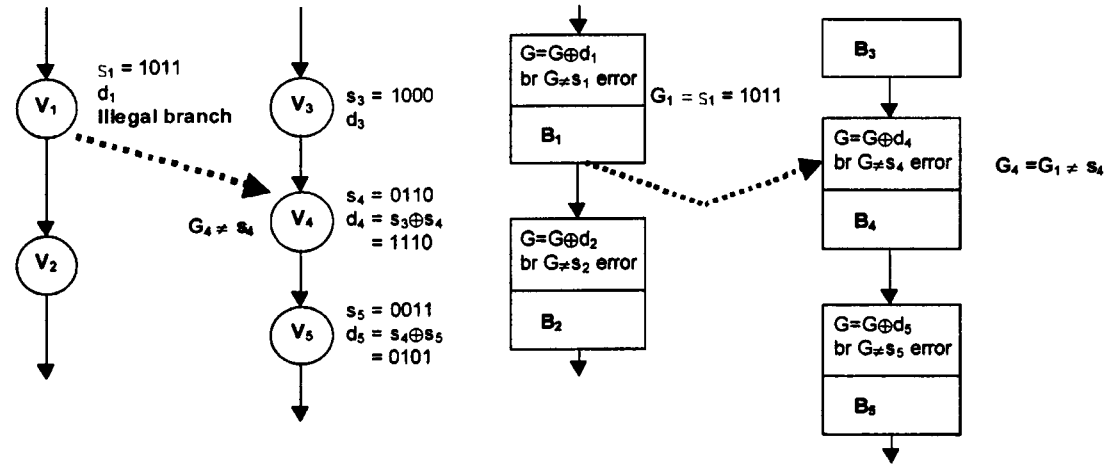


Fig. 7. Detection of a branch which is illegally jumping to instruction #2 of a basic block.

B. The Run-Time Adjusting Signature, D

Illegal branches that violate the CF can be detected by assigning unique signatures to each of the nodes in the program graph, and adding signature checking instructions to them. However, there are cases where the same signature has to be assigned to multiple nodes, e.g., a branch-fan-in node. In Fig. 8, the two nodes, v_1, v_3, have branches to the same node, a branch-fan-in v_5. As stated before, $d_5 = s_1 \oplus s_5$, and there is no problem when the $\text{br}_{1,5}$ is taken because

$$G_5 = G_1 \oplus d_5 = s_1 \oplus s_1 \oplus s_5 = s_5,$$

which is the signature of v_5. If the $\text{br}_{3,5}$ is taken, however, the run-time signature G at v_5 is not equal to s_5, because

$$G_5 = G_3 \oplus d_5 = s_3 \oplus s_1 \oplus s_5 \neq s_5,$$

if $s_3 \neq s_1$.

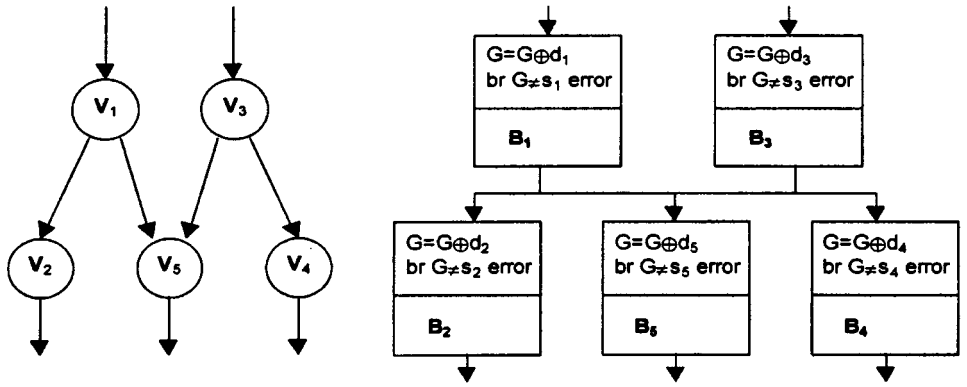


Fig. 8. Node v_1 and node v_3 have the same signatures.

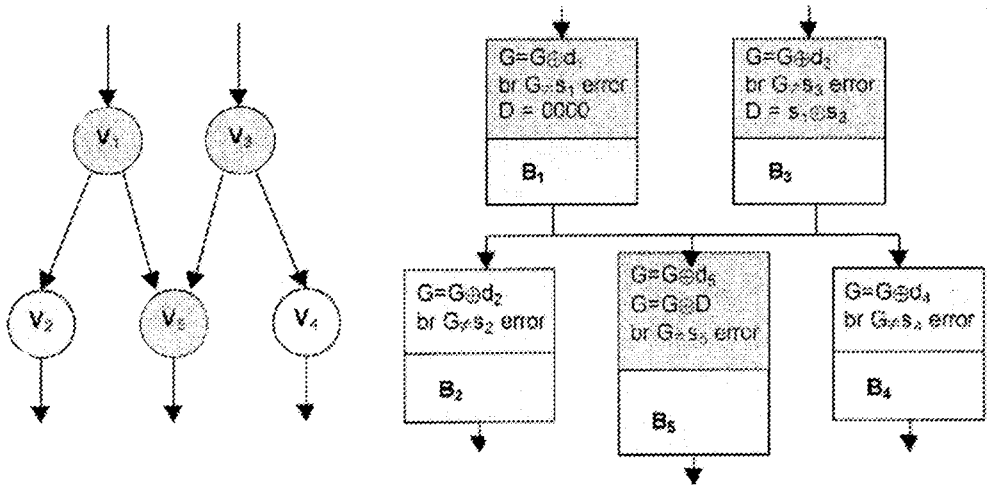


Fig. 9. Node v_1 and node v_3 have different signatures.

However, if $s_1 = s_3$ is used as the signatures, then an illegal branch from v_1 to v_4, or from v_3 to v_2, will not be detected. To solve the problem of assigning the same signature to multiple predecessors of a branch-fan-in node, a D is introduced. After the G is generated by the signature generation function, G is XOR'ed with D to get the signature of the branch-fan-in node; thus, at the source node, D has to be set to the value which makes G equal to the signature of the destination node.

Fig. 9 illustrates an example where D is used in the branch-fan-in node. At v_5, one more checking instruction $G = G \oplus D$ is added. After the $G = G \oplus d_5$, then G is XOR'ed with D that should be determined at the source v_1 and v_3. Because d_5 is initially set to the XOR-difference between s_1 and $s_5 (d_5 = s_1 \oplus s_5)$, when the $\mathrm{br}_{1,5}$ is taken, the updated G is already the same as s_5; there is no need to change G, thus D is set to 0 at v_1

$$(G_5 = G_5 \oplus D = s_5 \oplus 0000 = s_5).$$

When the $\mathrm{br}_{3,5}$ is taken, the updated G at the first line of v_5 is

$$G_5 = G_3 \oplus d_5 = s_3 \oplus (s_1 \oplus s_5).$$

To make G equal to s_5, the G should be XOR'ed with $s_1 \oplus s_3$ at the second line:

$$G = G_5 \oplus D = s_3 \oplus (s_1 \oplus s_5) \oplus (s_i \oplus s_3) = s_5.$$

Therefore, at the v_3, set $D = s_1 \oplus s_3$.

For the $\mathrm{br}_{1,2}$, the D is not necessary because v_2 is not a branch-fan-in node; only one branch is coming into v_2 and d_2 is equal to $s_1 \oplus s_2$. Thus, the updated G at v_2 is equal to s_2 as in the previous case in Fig. 8.

In summary, if a source node has a branch to a branch-fan-in node, the source node has to have one extra instruction for D in the checking instructions in order to set D to the appropriate value before branching. If the branch to the branch-fan-in node is taken, then D is XOR'ed with G at the destination node; otherwise, D is just ignored. In this way, arbitrary different numbers can be assigned to all the nodes in the program graph.

C. Algorithm A

Algorithm *A* assigns signatures to each node in a program flow-graph when a program is compiled.

$N \equiv$ total number of nodes in the program.

```
Algorithm A
1. Identify all basic blocks, build pro-
   gram flow-graph, and number all nodes in
   the program flow-graph.
2. Assign an s_i to v_i, in which s_i ≠ s_j if
   i ≠ j, i, j = 1, 2, ..., N.
3. For each v_j, j = 1, 2, ..., N.
```

```
  3.1 For v_j whose pred(v_j) is only one
  node v_i, then d_j = s_i ⊕ s_j.
  3.2 For v_j whose pred(v_j) is a set of
  nodes v_{i,1}, v_{i,2}, ..., v_{i,M}—therefore, v_j
  is a branch-fan-in node—the signature
  difference is determined by one of the
  nodes (picked arbitrarily) as d_j = s_{i,1}⊕s_j.
  For v_{i,m}, m = 1, 2, ..., M, insert an in-
  struction D_{i,m} = s_{i,1} ⊕ s_{i,m} into v_{i,m}. This
  instruction should be located after "br
  (G ≠ s_j) error" instruction in v_{i,m}.
  3.3 Insert an instruction G = G ⊕ d_j at
  the beginning of v_j.
  3.4 If v_j is a branch-fan-in node, then
  insert an instruction G = G ⊕ D after
  G = G ⊕ d_j in node v_j.
  3.5 Insert an instruction "br (G ≠ s_j)
  error" after the instructions placed in
  steps 3.3 or 3.4.
End-Algorithm
```

When a $\text{br}_{i,j}$ is taken, if the destination v_j is not a branch-fan-in node, then the run-time signature G_j is generated by the signature function $f(G_i, d_j) = G_i \oplus d_j$ and compared with the s_j of v_j. If they match, then no CF error has occurred in taking $\text{br}_{i,j}$.

When a $\text{br}_{i,j}$ is taken, if the destination v_j is a branch-fan-in node, then the run-time signature G_j is generated by the signature function and D:

$$G_j = f\left(f(G_i, d_j), D_j\right).$$

If they match, it means no CF error has occurred in taking $\text{br}_{i,j}$.

CFCSS detects the types of CF errors, as presented in corollaries 1–5.

Corollary 1: An illegal branch taken to the signature function instruction—the first line of the node—will be detected.

Proof: Suppose that $\text{br}_{i,j}$ is an illegal branch, then $v_i \notin \text{pred}(v_j)$. At v_i, $G = s_i$. After the branch is taken, the new run-time signature is generated,

$$G = G_j = G_i \oplus d_j = s_i \oplus s_k \oplus s_j;$$

s_k is the signature of $v_k = \text{pred}(v_j)$. Because s_i, s_k, s_j are all different numbers, then

$$G = s_i \oplus s_k \oplus s_j \neq s_j.$$

Mismatch occurs and the error is detected.

Corollary 2: An illegal branch taken to the instruction "br $(G \neq s)$ error" —the second line of the node—will be detected.

Proof: Suppose that $\text{br}_{i,j}$ is an illegal branch and the branch is taken to line #2 of the node, i.e., skipped the signature function. Because the new G was not generated, $G = s_i$, not s_j. Therefore, "br $(G \neq s)$ error" instruction sees the mismatch and detects the error.

Corollary 3: An illegal branch to the body of the node where the original basic block sits will be detected.

Proof: Suppose that $\text{br}_{i,j}$ is an illegal branch and the branch is taken to a place where one of the instructions in the original basic block is located, i.e., skipped the checking instructions and landed at one of the instructions in the original basic block. Because the new G is not generated at v_j, then $G = s_i$, not s_j although the control is transferred to the v_j. After the instructions in v_j are executed, then $\text{br}_{j,k}$ is taken, where $v_k \equiv \text{suc}(v_j)$. The checking instructions in v_k generate the updated

$$G = G_k = G_j \oplus d_k = G_i \oplus (s_j \oplus s_k) = s_i \oplus s_j \oplus s_k.$$

Because s_i, s_k, s_j are all different numbers, then

$$G = s_i \oplus s_k \oplus s_j \neq s_j.$$

Mismatch occurs and the error is detected.

Corollary 4: A branch insertion inside a node will be detected if it is an illegal branch.

Proof: Suppose that $\text{br}_{i,k}$ is inserted at v_i, then $\text{br}_{i,k} \notin E$ ($\text{br}_{i,k}$ is an illegal branch). At v_i, then $G = s_i$. After $\text{br}_{i,k}$ is taken to the first instruction of v_k, then the new updated G is

$$G = G_k = G_i \oplus d_k = s_i \oplus (s_k \oplus s_l)$$

in which s_l is the signature of v_l, and $v_l = \text{pred}(v_k)$. Because s_i, s_k, s_l are all different numbers, then

$$G = s_i \oplus s_k \oplus s_l \neq s_j$$

unless aliasing occurs (discussed in Section III-D). Mismatch occurs and the error is detected. By corollary 3, a branch to other instructions of the node will also be detected.

Corollary 5: The deletion of an unconditional branch instruction from the node will be detected.

Proof: Suppose that the branch instruction $\text{br}_{i,j}$ at v_i is changed to another instruction; then, $\text{br}_{i,j}$ is removed from E and an adjacent v_k is merged into v_i. Then, the signature of this node is changed from s_i to s_k in the middle of the node where v_i and v_k are merging; thus, the G should be updated to s_k. However, because $v_i \notin \text{pred}(v_k)$, then G will not match with s_k. Therefore, the error is detected. This is similar to the case where an illegal branch $\text{br}_{i,k} \notin E$ occurs.

D. Aliasing

If multiple nodes share multiple branch-fan-in nodes as their destination nodes, aliasing could occur between legal and illegal branches, and cause an undetectable CF error.

In Fig. 10, v_5 is a branch-fan-in node with 3 source-nodes v_1, v_2, v_3($\text{pred}(v_5) = \{v_1, v_2, v_3\}$). Node v_6 is also a branch-fan-in node but it has only 2 source-nodes v_2, v_3, not $v_i(\text{pred}(v_6) = v_2, v_3)$. According to algorithm *A*, step 3.4—first, the d_5 of v_5 is determined as $d_5 = s_2 \oplus s_5$. The runtime signature D_2 of v_2 is $D_2 = s_2 \oplus s_2 = 0000$, the $D_3 = s_2 \oplus s_3$, and $D_1 = s_2 \oplus s_1$. For the branch-fan-in v_6, the d_6 should also be calculated with $v_2(d_6 = s_2 \oplus s_6)$ because $\{v_2, v_3\}$ is a subset of both $\text{pred}(v_5)$ and $\text{pred}(v_6)$. In other words, either $d_6 = s_2 \oplus s_6$ or $d_6 = s_3 \oplus s_6$. However, because $\text{pred}(v_5) = \{v_1, v_2, v_3\}$ and $\text{pred}(v_6) = \{v_2, v_3\}$ have the same subset $\{v_2, v_3\}$, and d_5 of v_5 is already calculated with the signature of v_2, then d_6 of v_6 should also be calculated with the signature of v_2: $d_6 = s_2 \oplus s_6$. As a result, both v_5

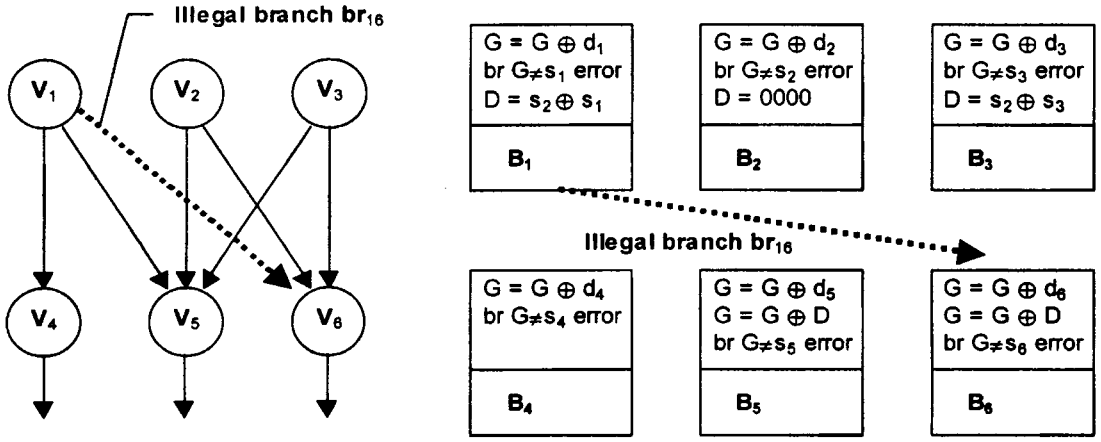


Fig. 10. Aliasing that causes an undetectable flow-error.

and v_6 have the difference signature calculated with the same $s_2(d_5 = s_2 \oplus s_5, d_6 = s_2 \oplus s_6)$.

Suppose that an illegal branch $\mathrm{br}_{1,6}$ occurs and lands at the first line of v_6, where the instruction of signature function $f(G_{\mathrm{prev}}, d_6)$ is located. G_{prev} is $G_1 = s_1$ and updated

$$G = G_6 = f(G_{\mathrm{prev}}, d_6) \oplus D = (G_1 \oplus d_6) \oplus (s_2 \oplus s_1)$$
$$= (s_1 \oplus (s_2 \oplus s_6)) \oplus (s_2 \oplus s_1) = s_6.$$

The updated $G_6 = s_6$; therefore, this illegal branch is not detected. This aliasing error is caused by the fact that more than 2 branch-fan-in nodes have their signature differences calculated with the signature of the same node, and their predecessor sets are not equal. More specifically, the condition for aliasing error is:

Aliasing error occurs when $d_i = s_s \oplus s_i$, $d_j = s_s \oplus s_j$, but pred$(v_i) \neq$ pred(v_j). If pred(v_i) − pred$(v_j) \neq \emptyset$, an illegal branch from a node in pred(v_i) − pred(v_j)—assuming pred$(v_j) \subset$ pred(v_i)—to v_j is undetectable when that branch is taken to the location of the instruction for the signature function.

If the illegal branch is taken to any location except "the first line of the node"—the instruction for the signature function—the CF error is detected because the new run-time signature associated with the destination node is not generated. In other words, the illegal branch is detected unless it lands at the first line of the destination node that satisfies the condition described in the previous paragraph. With this observation, one can avoid the undetectable illegal branch if signatures are assigned to the nodes in the following way.

If 1 bit-error is assumed, and the Hamming distance between the "addresses of the first instructions in v_5 and v_6" is greater than 1, then this undetectable illegal branch is avoided; 1 bit-error in the destination field of the branch instruction at v_1 cannot cause an illegal branch to the location of the first line of v_6. Similarly, if m-bit-error is assumed, and the addresses of the first instructions of all successor nodes are different by Hamming distance greater than m, then undetectable illegal branches caused by aliasing are avoided.

IV. Simulation Results

Seven benchmark programs are chosen for the experiment:

- LZW (compression);
- FFT (Fast Fourier Transform);
- Matrix multiplication;
- Quick sort;
- Insert sort;
- Hanoi;
- Shuffle.

First, the source files were compiled, and assembly codes were generated for MIPS architecture. One of the following faults was randomly applied to the assembly code:

- a branch deletion in which the branch instruction is replaced by a nop instruction,
- a branch creation in which an unconditional branch is randomly inserted into the program,
- a branch operand change where the immediate field of the instruction is corrupted.

Then, a machine code was generated by compiling the faulty assembly program and executed on an SGI Indigo that uses the R4400 MIPS processor. Table I shows the results of 500 iterations.

- Row #2: The numbers (incorrect result) indicate the fraction of faults that cause the programs to produce incorrect results that go undetected.
- Row #3: Erroneous result is "infinitely produced" because the fault creates an infinite loop in the program.
- Row #4: The processor does not respond to inputs, so the processor must be manually stopped.
- Row #5: The fraction of faults that are detected by the operating system of the machine. A segmentation fault and failed assertion are examples of faults detected by the operating system. Some branch faults might not affect the outputs of the program. For example, suppose the constant field of a conditional branch instruction is corrupted. If the condition is not satisfied, the branch instruction is never taken, and the fault in the constant field has no effect on the outputs.
- Row #6: These are faults that have no effect on the outputs.

TABLE I
RESULT OF BRANCH-FAULT INJECTION INTO ORIGINAL PROGRAMS

	LZW (comp.)	FFT	Matrix mult.	Quick sort	Insert sort	Hanoi	Shuffle
Incorrect result	21.0%	11.8%	26.6%	15.8%	26.2%	12.0%	22.8%
Infinite erroneous result	3.6%	16.8%	1.0%	0%	1.2%	2.0%	0%
Processor hung	1.6%	16.8%	6.2%	17.8%	12.2%	7.8%	12.6%
Detected by OS	57.4%	36.0%	55.0%	50.0%	50.4%	54.6%	50.4%
Correct result	16.4%	18.6%	11.2%	16.4%	10.0%	23.6%	14.2%
Total	100%	100%	100%	100%	100%	100%	100%
Incorrect output undetected	26.2%	45.4%	33.8%	33.6%	39.6%	1.8%	35.4%

TABLE II
RESULTS OF BRANCH-FAULT INJECTION INTO THE PROGRAMS WITH CFCSS

	LZW (comp.)	FFT	Matrix mult.	Quick sort	Insert sort	Hanoi	Shuffle
Detected by CFCSS	30.8%	34.4%	41.0%	28,8%	37.2%	34.6%	40.0%
Incorrect result	0.8%	0%	2.4%	0.6%	0.6%	0.0%	1.6%
Infinite erroneous result	0.2%	2.8%	0%	0%	0%	0.2%	0%
Processor hung	1.0%	1.4%	0.6%	2.8%	3.4%	1.6%	1.6%
Detected by OS	12.0%	13.6%	9.4%	17.4%	11.0%	8.0%	7.2%
Correct result	55.2%	47.8%	46.6%	50.4%	47.8%	55.6%	49.6%
Total	100%	100%	100%	100%	100%	100%	100%
Incorrect output undetected	2.0%	4.2%	3.0%	3.4%	4.0%	1.8%	3.2%

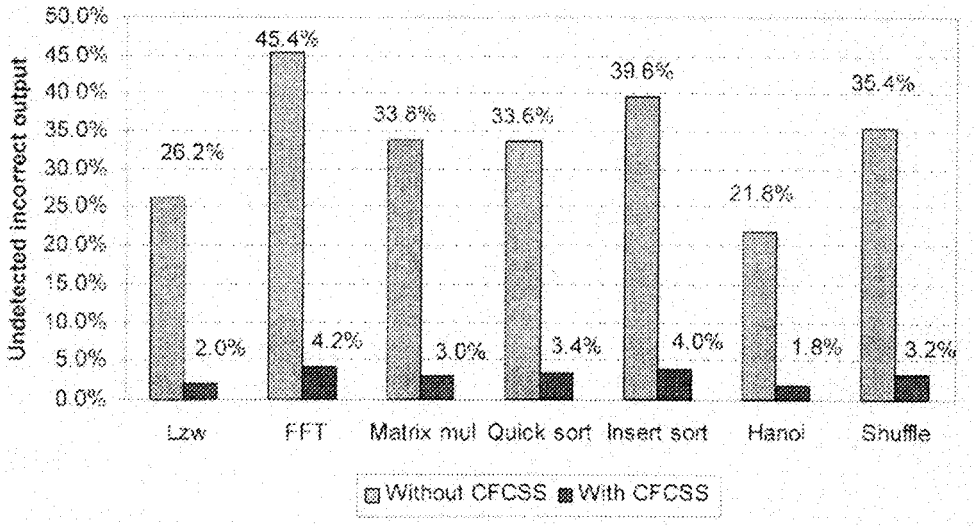


Fig. 11. Fraction of faults that produced undetected incorrect outputs.

- Row #7 (Last row): The fraction of faults that produced incorrect outputs without being detected: the sum of rows #2, #3, #4. On average, 33.7% of the injected faults are not fault-secure.

For part #2 of the experiment, CFCSS is included in the assembly source code, and the branch fault (branch deletion, branch creation, and operand change) is inserted into the code. The resulting assembly code is compiled and executed. Table II shows the result of 500 iterations; the last row is the fraction of faults that result in error and are not detected by either CFCSS or the operating system.

The graph in Fig. 11 illustrates the fraction of faults that are not detected for both the original program and the program with CFCSS. CFCSS shows high error detection capability: In the programs without CFCSS, an average of 33.7% of the injected branching faults produced undetected incorrect outputs; however, in the programs with CFCSS, only 3.1% of branching faults produced undetected incorrect outputs. The simulation results show that CFCSS increased the error detection capability by a factor of 10.

Table III shows the code-size overhead introduced by CFCSS in the benchmark programs that were used in the fault-injection simulations. Table III includes execution-time overhead for the benchmark programs. Column #2 gives the number of instructions before adding checking instructions to the original program. Column #3 gives the number of checking instructions added. Column #4 is the average size of basic blocks in the program. Columns #5 and #6 show the overhead of adding the checking instructions to the original program: the ratio of checking instructions to original instructions and execution time overhead. These numbers are based on the MIPS architecture where the "br $(G \neq s)$ error" is translated into two instructions: a compare-instruction followed by a branch-instruction. This might not be the case in other architectures; therefore, the numbers could be different—depending on processor architecture.

TABLE III
CHECKING INSTRUCTION OVERHEAD

	Number of —				
		Checking	Basic	Overhead	
Program	Instructions	Instructions	block size	Size	Exec. time
LZW (comp.)	452	280	4.3	61.9%	58.1%
FFT	411	140	8.7	34.0%	16.2%
Matrix mult.	763	316	7.4	41.4%	33.4%
Insert sort	132	84	4.5	63.6%	69.2%
Quick sort	254	142	5.4	55.9%	61.8%
Hanoi	179	60	6.6	33.5%	35.2%
Shuffle	688	183	11.7	26.6%	28.1%

Calculation-intensive programs, such as FFT, have larger basic blocks than data-analysis programs. Thus the overhead is smaller for calculation-intensive programs. On the other hand, programs such as sorting and searching have small basic blocks because they have frequent branch instructions; therefore, the overhead of checking instructions in these programs is higher compared to calculation-intensive programs.

V. OVERHEAD REDUCTION AND EXAMPLES

For the overhead of algorithm *A*, each node has between two to four additional instructions. If the average size of one basic block is seven to eight instructions [23], then the code size overhead is about 25% to 43%. Every node has one instruction that compares the run-time signature with the signature of that node. If it is not critical to detect an error immediately (if a longer latency for detecting an error is allowed) then comparing the signatures can be delayed. Once an illegal branch is taken, and the run-time signature becomes unequal to the signature of the node, then the run-time signature does not match the signatures of the following nodes in the CF because all signatures are different in the program graph. Therefore, the comparison instruction can be put only in some of the nodes and a mismatch of the signatures in those nodes can be detected only after an illegal branch. Algorithm *B* reduces overhead in algorithm *A* by postponing the comparisons. The difference between algorithms *A* and *B* is in step 3.5. Instead of comparing signatures at every node, algorithm *B* compares the signatures at the nodes where it is important to detect the CF errors.

$N \equiv$ total number of nodes in the program.

Algorithm *B*
1. Identify all basic blocks, build program flow graph, and number all nodes in the program flow graph.
2. Assign an s_i to v_i in which $s_i \neq s_j$ if $i \neq j$, i, $j = 1, 2, \ldots, N$.
3. For each v_j, $j = 1, 2, \ldots, N$.
   3.1 For v_j whose pred(v_j) is only 1 node, v_i, then $d_j = s_i \oplus s_j$.
   3.2 For v_j whose pred(v_j) is a set of nodes $\{v_{i,1}, v_{i,2}, \ldots, v_{i,M}\}$—therefore, v_j is a branch-fan-in node—the signature difference is determined by one of the nodes (picked arbitrarily) as $d_j = s_{i,1} \oplus s_j$. For node $v_{i,m}$, $m = 1, 2, \ldots, M$, insert a $D_{i,m} = s_{i,1} \oplus s_{i,m}$ into $v_{i,m}$. This instruction should be located after "br $(G \neq s_j)$ error" instruction in $v_{i,m}$.
   3.3 Insert a $G = G \oplus d_j$ at the beginning of v_j.
   3.4 If v_j is a branch-fan-in node, then insert an instruction $G = G \oplus D$ after $G = G \oplus d_j$ in v_j.
   3.5 Insert an instruction "br $(G \neq s_j)$ error" only into v_j where the comparison between the run-time signature $G = G_i$ and the signature s_i is wanted.
End_Algorithm

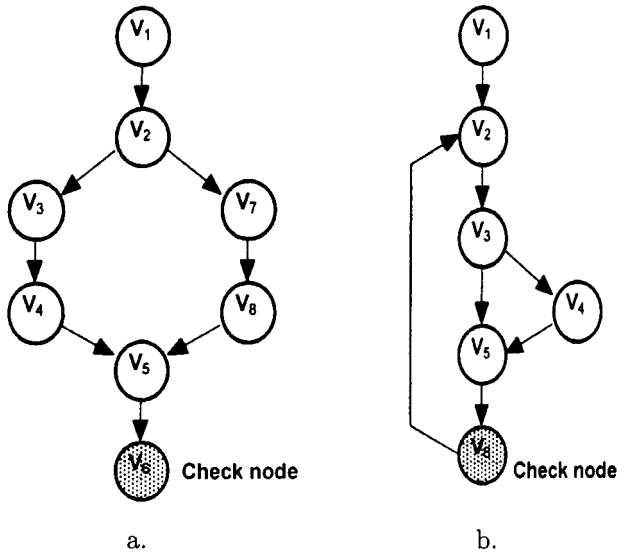


Fig. 12. Two examples of algorithm *B*.

Fig. 12 has two examples of algorithm *B*. In Fig. 12(a), the run-time signature and the node signature are compared only at v_6. The overhead for v_i, v_2, v_3, v_7 is only the $f(G, d)$. The branch-fan-in v_5 has two instructions for $f(f(G, d), D)$. Nodes v_4 and v_8 that are branching to the branch-fan-in v_5 have two instructions: one for signature function and the other for loading D. Algorithm *A* has an overhead of 30% in Fig. 12(a), assuming that all basic blocks have eight instructions. Using algorithm *B* for the same example, the overhead is reduced to 19%. However, the average detection latency in algorithm *B* is 22.8 instructions while that in algorithm *A* is 4.5 instructions.

Fig. 12(b) is an example of a loop in which the signature is checked only at the end of the loop instead of checking it at every

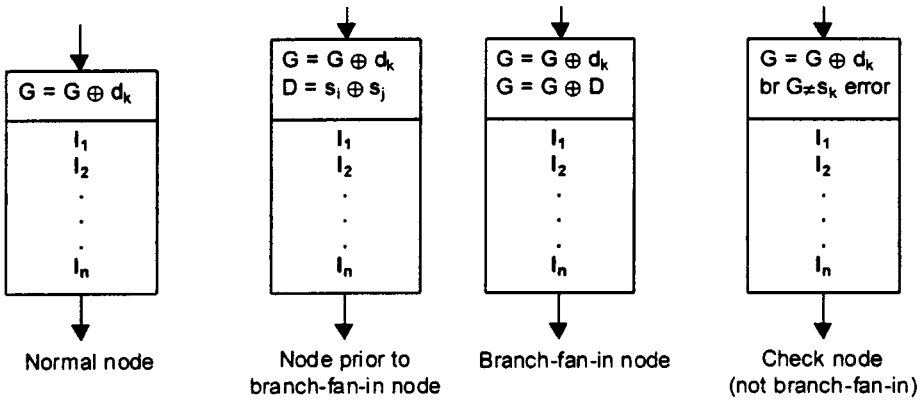


Fig. 13. Checking instructions in algorithm *B*.

node in the loop. Similar to the example in Fig. 12(a), only v_6 has "br $(G \neq s)$ error" instruction. The overhead of applying algorithm *A* to this loop example is 37.5% with the latency of 4.5 instructions. When algorithm *B* is applied, the overhead is reduced to 27%, but the average detection latency increases to 21.8 instructions.

Fig. 13 shows four different nodes with associated checking instructions. A check node is a node that has the comparison instruction "br $(G \neq s)$ error." The comparison instruction is excluded from the node that is not a check node. A node can be a combination of two types: a branch-fan-in and a check node.

ACKNOWLEDGMENT

The authors would like to thank Dr. N. Saxena for his valuable suggestions, and Dr. P. Hubbard and S. Mitra for their helpful comments.

REFERENCES

[1] D. J. Lu, "Watchdog processor and structural integrity checking," *IEEE Trans. Computers*, vol. C-31, pp. 681–685, July 1982.
[2] G. Miremadi, J. Karlsson, J. U. Gunneflo, and J. Torin, "Two software techniques for on-line error detection," in *Digest of Papers, 22nd Ann. Int. Symp. Fault-Tolerant Computing*, 1992, pp. 328–335.
[3] S. S. Yau and F.-C. Chen, "An approach to concurrent control flow checking," *IEEE Trans. Software Eng.*, vol. SE-6, Mar. 1980.
[4] A. Ersoz, D. M. Andrews, and E. J. McCluskey, "The watchdog task: Concurrent error detection using assertions," Center for Reliable Computing, Stanford Univ., CA, CRC-TR 85-8, 1985.
[5] M. Namjoo, "Techniques for concurrent testing of VLSI processor operation," in *Digest of Papers, IEEE Test Conf.*, 1982, pp. 461–468.
[6] J. P. Shen and M. A. Schuette, "On-line self-monitoring using signatured instruction streams," in *Int. Test Conf. Proc.*, 1982, pp. 275–282.
[7] J. B. Eifert and J. P. Shen, "Processor monitoring using asynchronous signatured instruction streams," in *Digest of Papers, 14th Ann. Int. Conf. Fault-Tolerant Computing*, 1984, pp. 394–399.
[8] K. Wilken and J. P. Shen, "Concurrent error detection using signature monitoring and encryption: Low-cost concurrent-detection of processor control errors," in *Dependable Computing for Critical Applications*, A. Avizienis and J. C. Laprie, Eds: Springer-Verlag, 1989, vol. 4, pp. 365–384.
[9] ——, "Continuous signature monitoring: Low-cost concurrent-detection of processor control errors," *IEEE Trans. Computer-Aided Design*, vol. 9, pp. 629–641, June 1990.
[10] N. R. Saxena and E. J. McCluskey, "Control-flow checking using watchdog assists and extended-precision checksums," *IEEE Trans. Computers*, vol. 39, pp. 554–559, Apr. 1990.
[11] H. Madeira and J. G. Silvia, "On-line signature learning and checking," in *Dependable Computing for Critical Applications 2*, J. F. Schlichting and R. D. Schlichting, Eds: Springer-Verlag, 1992, pp. 395–420.
[12] J. Ohlsson and M. Rimen, "Implicit signature checking," in *Digest of Papers, 25th Int. Symp. Fault-Tolerant Computing*, 1995, pp. 218–227.
[13] C. H. Tung and C. W. McCarron, "Concurrent control flow checking in sequential and parallel program," in *24th Asilomar Conf. Signals, Systems and Computers*: Maple, 1990, vol. 2, pp. 851–855.
[14] P. Furtado and H. Madeira, "Fault injection evaluation of assigned signatures in RISC processors," in *Proc. 2nd Euro. Dependable Computing Conf.*, 1996, pp. 55–72.
[15] P. P. Shirvani and E. J. McCluskey, "Fault-tolerant systems in a space environment: The CRC ARGOS project," Center for Reliable Computing, Stanford Univ., CA, CRC-TR 98-2, 1998.
[16] K. S. Wood *et al.*, "The USA experiment on the ARGOS satellite: A low cost instrument for timing x-ray binaries," in *EUV, X-Ray, and Gamma-Ray Instrumentation for Astronomy V SPIE Proc.*, vol. 2280, 1994, pp. 19–30.
[17] D. J. Lu, "Watchdog processors and VLSI," in *Proc. Nat. Electron. Conf.*, vol. 34, 1980, pp. 240–245.
[18] A. Mahmood and E. J. McCluskey, "Watchdog processor: Error coverage and overhead," in *Digest, 15th Ann. Int'l. Symp. Fault-Tolerant Computing (FTCS-15)*, 1985, pp. 214–219.
[19] D. Andrews, "Using executable assertions for testing and fault tolerance," in *9th Fault-Tolerance Computing Symp.*, 1979, pp. 20–22.
[20] H. Madeira, M. Rela, and J. G. Silvia, "Time behavior monitoring as an error detection mechanism," in *Dependable Computing for Critical Applications 2*, J. F. Schlichting and R. D. Schlichting, Eds: Springer-Verlag, 1993, pp. 395–420.
[21] M. A. Shuette and J. P. Shen, "Exploiting instruction-level parallelism for integrated control-flow monitoring," *IEEE Trans. Computers*, vol. 43, pp. 129–140, Feb. 1994.
[22] V. V. Ignatushchenko *et al.*, "Effectiveness of temporal redundancy of parallel computational processes," *Automation and Remote Control*, pt. 2, vol. 55, no. 6, pp. 900–911, June 1994.
[23] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 2nd ed., 1996.

Nahmsuk Oh received the B.E. degree (1996) in electronics engineering from Yonsei University, Korea; the M.S.E.E. degree (1998), the Ph.D. degree in electrical engineering and the Ph.D. minor in computer science (2001), all from Stanford University.

He is currently with Synopsys Inc., Mountain View, CA as a Senior R&D Engineer in the Test Automation Products Group. His research interests include fault-tolerant computing, computer architecture, logic synthesis, and VLSI design/test.

Dr. Oh is a member of the IEEE and the Computer Society.

Philip P. Shirvani received the B.S. degree (1991) in electrical engineering from Sharif University of Technology, Tehran, Iran; and the M.S. (1996) and Ph.D. degrees in electrical engineering from Stanford University.

He is an Assistant Director for the Stanford ARGOS Project at Center for Reliable Computing, Stanford University. His research interests include computer architecture, fault-tolerant computing, and VLSI design and test. He

Dr. Shirvani is a member of the IEEE and the Computer Society.

Edward J. McCluskey received the A.B. (*summa cum laude*, 1953) in mathematics and physics from Bowdoin College, and the B.S. (1953), M.S. (1953), and Sc.D. (1956) degrees in electrical engineering from MIT. The degree of Doctor Honoris Causa (1994) was awarded by the Institut National Polytechnique de Grenoble.

He worked on electronic switching systems at the Bell Telephone Laboratories in 1955–1959. In 1959, he moved to Princeton University, where he was Professor of Electrical Engineering and Director of the University Computer Center. In 1966, he joined Stanford University, where he is a Professor of Electrical Engineering and Computer Science, and Director of the Center for Reliable Computing. He founded the Stanford Digital Systems Laboratory (now the Computer Systems Laboratory) in 1969, and the Stanford Computer Engineering Program (now the Computer Science M.S. Degree Program) in 1970. The Stanford Computer Forum (an Industrial Affiliates Program) was started by Dr. McCluskey and 2 colleagues in 1970; he was the Director until 1978. He developed the first algorithm for designing combinational circuits—the Quine–McCluskey logic minimization procedure as a doctoral student at MIT. At Bell Labs and Princeton, he developed the modern theory of transients (hazards) in logic networks and formulated the concept of operating modes of sequential circuits. His Stanford research focuses on logic testing, synthesis, design for testability, and fault-tolerant computing. Prof. McCluskey and his students at CRC worked out many key ideas for fault-equivalence, probabilistic modeling of logic networks, pseudo-exhaustive testing, and watchdog processors. He collaborated with Signetics researchers in developing one of the first practical multivalued logic implementations and then worked out a design technique for such circuitry. He has published several books and book chapters, as well as hundreds of papers. His most recent book is *Logic Design Principles With Emphasis on Testable Semicustom Circuits*, (Englewood Cliffs, NJ: Prentice-Hall, 1986).

Dr. McCluskey served as the first President of the IEEE Computer Society. His most recent honors include election (1998) to the National Academy of Engineering, the 1996 IEEE Emanuel R. Piore Award, and IEEE Computer Society Golden Core Member. He received the IEEE Centennial Medal (1984) and the IEEE Computer Society Technical Achievement Award in Testing (1984). He received the EURO ASIC 90 Prize for Fundamental Outstanding Contribution to Logic Synthesis (1990). The IEEE Computer Society honored him with the 1991 Taylor L. Booth Education Award. He is a Fellow of the IEEE, AAAS, and ACM.