Chinese Journal of Aeronautics, 2013,26(2): 413–422

Chinese Society of Aeronautics and Astronautics
& Beihang University

Chinese Journal of Aeronautics

cja@buaa.edu.cn
www.sciencedirect.com

Graph-tree-based software control flow checking for COTS processors on pico-satellites

Yang Mu [a], Wang Hao [a,*], Zheng Yangming [b], Jin Zhonghe [a]

[a] *Department of Information Science and Electronic Engineering, Zhejiang University, Hangzhou 310027, China*
[b] *School of Aeronautics and Astronautics, Zhejiang University, Hangzhou 310027, China*

Received 2 November 2011; revised 25 November 2011; accepted 4 January 2012
Available online 7 March 2013

KEYWORDS

Branching error;
Commercial-off-the-shelf (COTS);
Control flow checking;
Error injection;
Graph tree;
On-board computer;
Pico-satellite

Abstract This paper proposes a generic high-performance and low-time-overhead software control flow checking solution, graph-tree-based control flow checking (GTCFC) for space-borne commercial-off-the-shelf (COTS) processors. A graph tree data structure with a topology similar to common trees is introduced to transform the control flow graphs of target programs. This together with design of IDs and signatures of its vertices and edges allows for an easy check of legality of actual branching during target program execution. As a result, the algorithm not only is capable of detecting all single and multiple branching errors with low latency and time overheads along with a linear-complexity space overhead, but also remains generic among arbitrary instruction sets and independent of any specific hardware. Tests of the algorithm using a COTS-processor-based on-board computer (OBC) of in-service ZDPS-1A pico-satellite products show that GTCFC can detect over 90% of the randomly injected and all-pattern-covering branching errors for different types of target programs, with performance and overheads consistent with the theoretical analysis; and beats well-established preeminent control flow checking algorithms in these dimensions. Furthermore, it is validated that GTCGC not only can be accommodated in pico-satellites conveniently with still sufficient system margins left, but also has the ability to minimize the risk of control flow errors being undetected in their space missions. Therefore, due to its effectiveness, efficiency, and compatibility, the GTCFC solution is ready for applications on COTS processors on pico-satellites in their real space missions.

1. Introduction

Single event upset (SEU) was first discovered in 1962.[1] Since then, 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.[2] Soft errors may be explicit bit flips in latches or memories, or glitches in combinational logics that can propagate and be captured in latches.[3] One of the major consequent problems is control flow errors in programs running on the computers. If not handled

* Corresponding author. Tel.: +86 571 87953857.
E-mail addresses: alexander_young@zju.edu.cn (M. Yang), roger@zju.edu.cn (H. Wang).

Peer review under responsibility of Editorial Committe of CJA.

Production and hosting by Elsevier

1000-9361 © 2013 Production and hosting by Elsevier Ltd. on behalf of CSAA & BUAA. Open access under CC BY-NC-ND license.
http://dx.doi.org/10.1016/j.cja.2013.02.019

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. This is especially true for the commercial-off-the-shelf (COTS) processors without radiation hardening, whose space application is witnessing rapid growth nowadays.[4]

Researchers have reported various attempts to solve the problem. Almost all of them are based on the idea of partitioning target programs into blocks with no branch instructions other than the last one (basic blocks), acquiring characteristics of legal control flows (usually signatures of basic blocks and branching among them), and carrying out control flow legitimacy checking during target program execution via comparison of such characteristics to those of the runtime branching. According to their resorts, the solutions fall into two categories, namely hardware-assisted and software.

Hardware-assisted solutions supervise program control flows making use of dedicated external devices like watchdog processors to target processors running the programs. Such devices calculate and pre-store signatures of legal branching out of binary codes or memory addresses of related instructions, and compare them to their regenerated versions at runtime. For example, control flow checking by execution tracing (CFCET)[5] employed execution tracing features within COTS processors and an external watchdog processor to monitor addresses of branches taken in programs. A watchdog processor monitoring scheme[6] based on discrete cosine transformation was also developed for very long instruction word length processors. Meanwhile, Michel et al.[7] went even further to build two levels of control flow checking capabilities within application-specific microprocessors.

On the other hand, software solutions turn to extra embedded checking instructions in target programs instead of dedicated hardware, so as to have the target processors do checking jobs themselves. Specifically, there are several ways to do so:

(1) Assign signatures to basic blocks and regenerate them during program execution for comparison to perform branching legality checking. Control-flow checking by software signatures (CFCSS)[8] assigned and embedded a unique signature in each basic block, regenerated it deductively at runtime, and revealed control flow errors through comparing the two. Control-flow checking using assertions (CCA)[9] identified branch-free intervals of target programs, fortified their entry and exit points through pre-inserted assertions, and carried out the checking via similar comparison. Enhanced control-flow checking with assertions (ECCA)[9] assigned signatures making use of prime number characteristics, and exposed control flow errors with divide-by-zero exceptions of target processors. Some other examples are: structural integrity checking (SIC)[10], block signature self-checking[11], and versatile assign signature checking (VASC).[12]
(2) Derive basic block signatures from their traits like instruction codes and memory addresses, and implement checking by means of comparison similar to that described in Way (1). Examples include continuous signature monitoring (CSM)[13], on-line signature learning and checking (OSLC)[14], region-based control flow checking (RCF)[14], edge control flow checking technique (EdgCF)[15], and control-flow checking in the end (CFC-End).[16]
(3) Track program control flow path and check its legitimacy. Yau and Chen[17] introduced a control flow checking scheme that stored information of all possible control flow paths and checked the runtime path against it. Meanwhile, control-flow checking via regular expressions[18] assigned each basic block a unique string, acquired a regular expression for all possible string compositions, and checked the consistency of that of the runtime path with it for errors.

Existent hardware-assisted and software solutions have shown apparent shortcomings that lower their values of application. The formers called for specific extra hardware for control flow checking, so they had hardly any compatibility among different processors, although they usually required far less time and memory overheads than their software counterparts or none at all.[5–7,19] As for the latters, most of them could not detect all branching errors, especially the multiple ones. While among the few that could, most employed sophisticated mathematics. Such strategies imposed either enormous calculation overhead, or use of specific hardware like multipliers[9], also sacrificing the generality of the solutions. In addition, it has become a mainstream for pico- and other modern small (including micro- and nano-) satellite engineering to resort to a great variety of COTS processors as major components of on-board computers (OBCs), thanks to their costs and attainability.[4,20,21] Therefore, the demand for generic software control flow checking solutions for space-borne COTS processors has been craved.

In fulfilling such a demand, this paper proposes a generic high-performance and low-time-overhead software solution for control flow checking, graph-tree-based control flow checking (GTCFC). GTCFC works out the problem by transforming control flow graph of programs into an equivalent graph tree notation with an exclusive-parent topology similar to common trees. By doing this, control flow checking is also transformed into checking whether each vertex or virtual vertex is arrived at via its exclusive parent. GTCFC partitions a target program into basic blocks and builds its control flow graph as most algorithms.[8,9] Instead of applying to conventional graph storage schemes, the new graph tree data structure is introduced to transform the control flow graph. A graph tree has each of its (virtual) vertices have only one parent as stated above, so it can be stored with a linear space complexity. IDs of graph tree (virtual) vertices and signatures of control flow branches are correspondently designed to allow for easy regeneration and comparison. Along with the execution of the target program, the runtime signatures are generated at the entrance and the exit, and checked at the exit of each basic block. As a result, GTCFC not only has the capability of detecting all single and multiple branching errors with low latency and time overheads along with a linear-complexity memory overhead, but also remains generic among arbitrary instruction sets and independent of any specific hardware. Moreover, it can be easily accommodated into COTS processors and ease threats posed by SEU-induced control flow errors to a minimal level throughout life expectancies of common pico-satellites. Therefore, it is applicable to and useful for such processors on pico-satellites during their space missions.

2. Theory and realization

GTCFC transforms control flow graph of a target program into its graph tree notation that can be stored with a linear space complexity, assigns each (virtual) vertex in the graph tree (denoting a basic block in the program) a specific ID and each edge a signature, and performs checking of branching by comparing the pre-stored signatures to their runtime-generated counterparts. This section gives the details of the algorithm.

2.1. Definition

To start with, definitions of relevant concepts (partly inherited from Oh et al.[8]) are provided, despite that some of them have already been referred to as follows.

Basic block: a maximal set of ordered instructions that executes from the first to the last, with no branching except possibly the last. Vertex: basic blocks are denoted by vertices v_i ($i \in \{1, 2, \ldots, N\}$) in the control flow graph G, where N is the total number of basic blocks. V: $\{v_i; i \in \{1, 2, \ldots, N\}\}$, a set of vertices denoting basic blocks. suc(v_i): a set of successors of v_i in G. pred(v_i): a set of predecessors of v_i in G. Virtual vertex: $vv_{i,p}$ ($i \in \{1, 2, \ldots, N\}$, $p \in \{1,2, \ldots, P_i\}$, card(pred($v_i$)) $\geqslant$ 1), where card(pred(v_i)) is thenumber of the elements of the set pred(v_i) and P_i the number of virtual vertices divided from v_i; logical mirrors of divided vertices for use in building the graph tree notation of the control flow graph gtG. VV: $V \cup \{vv_{i,p}; i \in \{1, 2, \ldots, N\}, p \in \{1,2, \ldots, P_i\}$, card(pred($v_i$))$\geqslant$1$\}$, a set of vertices and virtual vertices in gtG. Edge: a legal branching from one basic block to another is denoted by a directed edge between the two correspondent (virtual) vertices in G. E: $\{e_{i,p;\ j,q}; i, j \in \{1, 2, \ldots, N\}, p \in \{1,2, \ldots, P_i\}, q \in \{1,2, \ldots, P_j\}, v_j \in \text{pred}(v_i)\}$, a set of edges denoting legal branches. Virtual edge: $ve_{i,p}$ ($i \in \{1, 2, \ldots, N\}$, $p \in \{2, 3, \ldots, P_i\}$), circuits in G are substituted by virtual edges in gtG. VE: $E \cup \{ve_{i,p}; i \in \{1, 2, \ldots, N\}, p \in \{2, 3, \ldots, P_i\}\}$, a set of edges and virtual edges in gtG. G: control flow graph $\{V, E\}$. From the definitions of V and E, a program can be represented by a control flow graph, G. gtG: the graph tree notation of the control flow graph {VV, VE}. aID_v_i ($i \in \{1, 2, \ldots, N\}$): assigned ID of v_i. aID_$vv_{i,p}$ ($i \in \{1, 2, \ldots, N\}$, $p \in \{2, 3, \ldots, P_i\}$): assigned ID of $vv_{i,p}$. s_$e_{i,p;\ j,q}$: signature of $e_{i,p;j,q}$. $br_{i,p;j,q}$: a branch from one (virtual) vertex to another. rs_$br_{i,p;\ j,q}$: runtime signature of a branch from one (virtual) vertex to another. Branching error/illegal branching: v_j is in suc(v_i) and v_i is in pred(v_j) if and only if $br_{i,j}$ is included in E. If a program is represented by its $G = \{V, E\}$, $br_{i,j}$ is illegal if it is not included in E. Illegal branches can result in various errors as stated above.

2.2. Graph tree

Control flow graphs are transformed into their graph tree notations in order to allow each (virtual) vertex an exclusive parent as in common trees. The fundamental idea of this transformation is to divide a vertex with multiple predecessors into equally multiple virtual vertices. Fig. 1 shows an example of a control flow graph and the result of the transformation, or its graph tree notation gtG, in its left and right halves, respectively. In the figure, the solid circles represent the vertices in both G and gtG, while the dashed circles represent the virtual vertices in gtG. Similarly, the solid and dashed arrows represent the edges in both G and gtG, and the virtual edges in gtG, respectively.

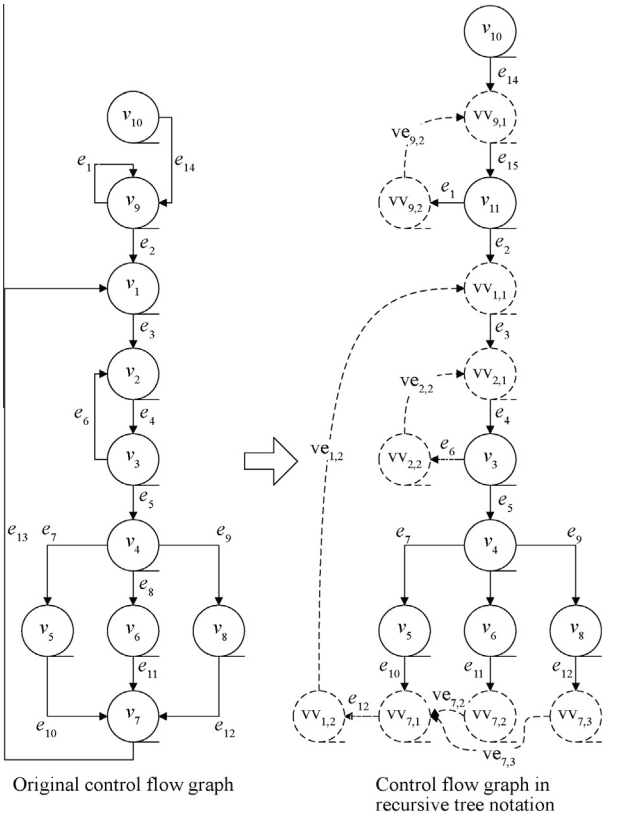

Fig. 1 Generation of a graph tree.

Theorem 1. A control flow graph G can be transformed into a notation where each vertex bears an exclusive predecessor (the graph-tree notation).

Proof. A vertex within a graph has to bear one of the four patterns of in-degrees, namely with no, a single, or multiple predecessors, and a loop. With the process stated below, all four types of vertices either have only one predecessor in the first place, or can be transformed into a set of (virtual) vertices where each does so. As a result, the process yields a notation of the original graph where each vertex bears an exclusive predecessor. Such a notation of the graph is its graph-tree notation.

At compilation time, the control flow graph of the target program is acquired as in most control flow checking algorithms, and the following process is performed to obtain its graph tree notation gtG.

(1) Vertices with no or one predecessor remain unchanged. Examples are v_{10}, v_3, v_4, v_5, v_6, and v_8 in Fig. 1.
(2) Vertices with more than one predecessor are divided into the same number of virtual vertices as that of its predecessors. For convenience, virtual vertices of one vertex are always labeled in the ascending sequence of integers, and in correspondence to the same order of the labels of its predecessors. Each virtual vertex then has one of the

predecessors of its original vertex as its exclusive parent, while those labeled "1" additionally inherit the relationship with the successors of their original vertices. Bear in mind that the virtual vertices are logical mirrors existent only in gtG, instead of physical copies of the vertices. For example, v_7 in Fig. 1 bears three predecessors, thus are divided into three virtual vertices, $vv_{7,1}$ through $vv_{7,3}$. Each of the virtual vertices inherits a parent of the original vertex v_7 as its exclusive parent, i.e., v_5, v_6, and v_8, respectively. Additionally, $vv_{7,1}$ branches to $vv_{1,2}$ as a heritage of the edge from v_7 to v_1.

(3) Vertices with loops, i.e., edges pointing from the vertices to themselves, cannot be directly transformed in the same manner as Process (2), otherwise the loop would still be present in gtG to damage the tree nature. Thus for each vertex with a loop, a null vertex filled with one NOP and one branching to the entrance of the original vertex instructions is added upon compilation. Meanwhile, the original vertex need be altered to branch to the entrance of the null one with an extra edge. Such a pre-processing changes the loop into a circuit while retaining the original actual control flow. Afterwards, the graph tree notation generation for that vertex is the same as for others. For example, in Fig. 1, the generation process of the gtG has a null vertex v_{11} inserted between the vertex v_9 (with a loop) and its successor v_1, and v_{11} in turn branches back to v_9. Then v_9 is divided into two virtual vertices as Process (2) does.

(4) Virtual edges connecting virtual vertices are introduced into gtG to resemble circuits among vertices in G. Each virtual vertex but the one labeled "1" (for convenience) divided from a vertex bears one virtual edge towards the latter. Virtual edges indicate that as execution of their origins completes, control flow continues from their terminals. In other words, they are indicators of the logical-mirror relationship between virtual vertices, instead of true edges in trees. Therefore, they do not damage the tree nature that every (virtual) vertex bears only one parent. For example, in gtG of Fig. 1, virtual vertices $vv_{2,2}$, $vv_{7,2}$, and $vv_{9,2}$ have virtual edges pointing to $vv_{2,1}$, $vv_{7,1}$, and $vv_{9,1}$, respectively.

As a result, the graph tree notation of the control flow graph takes on the topology that every (virtual) vertex has only one parent.

2.3. (Virtual) vertex IDs and branching signatures

To each undivided vertex and virtual vertex labeled "1", a unique integer between 1 and card(V) is assigned as its ID (aID_v_i or aID_$vv_{i,1}$), which can also be considered as IDs of the basic blocks. Similarly, to each virtual vertex with a label other than 1, a unique integer between card(V) + 1 and card(VV) is assigned as its ID (aID_$vv_{i,p}$, $p > 1$). Thus a unique signature (s_$e_{i,p;j,q}$) composed of the IDs of the (virtual) vertices on both of its ends for each edge (excluding virtual edges) is also born. Note that for faster memory access of runtime signature checking, it is recommended that the length of the IDs be shorter than or equal to the word length W of the target processor. However, under circumstances of card(VV) larger than 2^W, it is necessary and applicable to apply IDs longer than W.

2.4. Control flow checking

GTCFC makes use of the graph tree data structure established above by checking whether a (virtual) vertex is arrived at via its exclusive parent. It generates the runtime edge signature at both the entrance and the exit of each basic block, and compares it to its pre-stored counterpart at the exit for control flow legality check.

IDs of (virtual) vertices and the relativity information of parents (RIP) are generated and pre-stored as constants and an array of (card(VV) + 1) constant elements respectively in the target program, along with the generation of the graph tree during the compilation of the target program. For RIP, IDs of (virtual) vertices are indices and IDs of their exclusive parents are elements; ID of the parent of the root (virtual) vertex of gtG is defined as 0; and a zero-indexed element is reserved for checking for special types of branching errors.

Two variables cVID and pVID for caching IDs of the current and previous vertices respectively are defined. These variables are meant to keep track of the last branching. An array of (card(V) + 1) elements cVVID for current virtual vertex IDs is also defined, with IDs of undivided vertices and virtual vertices labeled "1" being indices, and elements filled dynamically with IDs of the current virtual vertex along with the program execution. The zero-indexed element is left void. The introduction of this array aims at preserving the information of the origin and destination of the expected branching right before it takes place, and having the correspondent information of the branching that actually has taken place checked against the former for its correctness.

With such preparations, control flow checking is carried out in four steps at the entrances and exits of the basic blocks, as described below.

Step 1 Initialize the correspondent element of cVVID to the root (virtual) vertex of gtG as its ID, and the rest of its elements, cVID and pVID as 0.

Step 2 At the entrance of each basic block, update pVID with cVID, and cVID with the ID of the current vertex. This step is to identify the branching to the basic block (s_$e_{\text{pVID;cVVID[aID_}v_i\text{]}}$, symbolizing the current basic block or vertex with v_i) with its signature for the checking in Step 3, and preserve the ID of the current vertex for its successors.

Step 3 Before branching to one of its successors at the exit of each basic block, check whether RIP[cVVID[aID_v_i]], which is the ID of the exclusive parent of the current (virtual) vertex, is equal to pVID. That is, check whether $\text{rs\_br}_{\text{pVID;cVVID[cVID]}}$ is equal to s_$e_{\text{pVID;cVVID[aID_}v_i\text{]}}$. A non-equality result suggests that a control flow error has occurred, and the program execution has to be terminated for error handling.

Step 4 At the exit of each basic block, a conditional branching (if there is) has to be altered into the pattern of decision of branching, or the succeeding vertex and virtual vertex, followed by clearing cVVID[aID_v_i], updating the correspondent element to the succeeding vertex of cVVID with the ID of the decided succeeding virtual vertex, and the correspondent unconditional branching consecutively. This step is to identify the branching expected to take place next with its runtime signature ($\text{rs\_br}_{\text{pVID;cVVID[cVID]}}$).

Or in C-Language pseudo-code, the checking scheme can be described as follows:

```
{
...
unsigned int cVID = 0, pVID = 0;
unsigned int cVVID[card(V) +1] = {The element for the entrance virtual vertex equals to its ID, and all other equal to 0};
unsigned int const RIP[card(VV) + 1] = {Each element equals to the assigned ID of its exclusive parent};
...
void func(){
  ...
  // basic block i (v_i in G)
  // R1: Current Runtime Signature Generation
  L1: pVID = cVID;
  L2: cVID = aID_v_i;
  // R2: Original Codes
    Original codes except those branching to the succeeding basic block
  // R3: Runtime Signature Checking
L3: if (RIP[cVVID[aID_vi]] ! = pVID) goto ERROR_HANDLING;
  // R4: Branching Decision (if there is)
    Branching decision
  // R5: Succeeding Runtime Signature Generation
L4: cVVID[aID_v_i] = 0;
L5: cVVID[ID of the succeeding vertex] = ID of the succeeding virtual vertex;
  // R6: Branching:
    Branching to the succeeding basic block
  // End of basic block i (v_i in G)
  ...
}
...
ERROR_HANDLING:
...
}
```

The pseudo-code of the hardened basic block $i(v_i)$ is partitioned into six regions. R1 is for generating the runtime signature of the branching that has taken place right before its execution. R2 includes the original codes of the basic block except the branching decision instructions. R3 checks the runtime signature of the last branching (before the execution of R1). R4 includes the altered codes of the basic block for branching decision (if there is). R5 generates the runtime signature of the branching about to be made by the basic block. Note that if the branching from the basic block is unconditional, R4 is vacant and R5 follows R3 directly up, and then R6 kicks out the branching. Meanwhile, the inserted lines for control flow checking are labeled L1 through L5.

It should be added that if a checking line of Step 3 (L3) is inserted after Step 2 (R1) too, the detection latency could be understandably shortened due to the control flow checking in advance. This would be accompanied by the instruction and execution time overheads being magnified since more redundancy has been introduced, and the detection capability remaining the same. Thus, it is reasonable not to make such an insertion.

3. Detection capability

Before proving the detection capability of GTCFC, several assumptions have to be made. Data errors are assumed to be handled by other schemes. This can be paraphrased into that all data should be correct. Moreover, since branching errors within one basic block and incorrect conditional branching decisions can also be perceived as data errors[9], only cross-boundary errors are taken into consideration here.

If a branching from v_i to v_j is legal, i.e., it originates from the last instruction of basic block $i(v_i)$ and terminates at the first of basic block $j(v_j)$, and the latter is a legitimate successor of the former, pVID equals to aID_v_i, cVVID[aID_v_j] equals to the ID of the virtual vertex divided from v_j as the child of v_i, and RIP[cVVID[aID_v_j]] equals to pVID at L3 of v_j. Thus the branching is checked to be legal. Otherwise, the branching is illegal.

In the last section, a hardened basic block is partitioned into six regions. Correspondently, branching errors can be classified according to their different original and terminal regions within basic blocks, and the detection capability of GTCFC for cross-boundary branching errors can be proven via enumerating their combinations.

Theorem 2. GTCFC is able to detect all single and multiple cross-boundary branching errors.

Proof. Suppose basic block $i(v_i)$ erroneously branches to basic block $j(v_j)$, and basic block j is arrived at via a presumably legal branching (otherwise, it would have been detected in the manner proven below) from basic block $k(v_k)$. If the error is:

(1) From regions before L5 of v_i to regions before L4 of v_j: since L5 of v_i has not been executed, cVVID[aID_v_j] remains 0 at L3 of v_j. This makes RIP[cVVID[aID_v_j]] 0 and not equal to pVID. Therefore, the branching error from v_i to v_j is detected at L3 of v_j.
(2) From L5 of v_i to L1 of v_j: L5 of v_i has updated the correspondent element to the succeeding vertex of cVVID with the ID of the succeeding basic block. If v_j is the expected successor of v_i, the branching error only skips the branching instruction from v_i to v_j, so no control flow error occurs in reality. If it is not, cVVID[aID_v_j] remains 0. This allows the error to be detected at L3 of v_j in a similar manner to (1).
(3) From L5 of v_i to L2 of v_j: at L3 of v_j, pVID remains aID_v_k, while cVVID[aID_v_j] has been updated to the ID of the succeeding virtual vertex of v_i. This results in an inequality between RIP[cVVID[aID_v_j]] and pVID. Thus the error is detected at L3 of v_j.
(4) From L5 of v_i to L3 of v_j: this type of branching errors bears the same pVID and cVVID[aID_v_j] values as in (2). As a result, the error is detected at L3 of v_j.
(5) From all regions of v_i to regions of v_j after L3: as the execution of the remainder of v_j results in a branching to the runtime successor of v_j, such errors can be perceived as ones from v_i to it. Therefore, they can be detected at L3 of the runtime successor of v_j as explained above.
(6) In conclusion, GTCFC has the capability of detecting all single (categories (1) through (4)) and multiple (Category (5)) cross-boundary branching errors.

4. Performance and overheads analysis

The performance and overheads of certain algorithms can be variable while retaining similar traits and relativities among different architectures and instruction sets. Thus, although the analysis for performance (detection latency) and overheads (memory, instruction, and execution time) of GTCFC has to be instantiated, it is logically significant in a general sense. Here and later, the analysis and tests are based on the realization of the algorithm on OBCs of in-service ZDPS-1A pico-satellites and their proposed successive products.[22] The OBC is mainly responsible for handling (including acquisition, formatting, storage, and distribution) of onboard data of its host satellite. It utilizes an 8052-compatible COTS processor, ADuC841 from Analog Devices. The processor possesses a Princeton architecture, with an on-chip random access memory (RAM) for data of 2 kbytes (including its register file), and a non-volatile floating-gate read-only memory (ROM) for programs of 62 kbytes. Like most micro-controllers, its program is executed directly on the ROM, without having to be loaded into a program RAM. The processor runs at a core frequency of 4 MHz. Yet unlike most 8052-compatible ones that take 12 clock cycles to accomplish an instruction cycle, it executes one instruction per clock cycle. That is to say, it executes roughly 4 million instructions per second (MIPS). Additionally, the processor carries peripherals of a timer and a serial port as included in standard 8052 processors. As seen later, they allow for a great convenience for the tests to be described in Section 5.

The platform for instantiating the solution is chosen based on the following reasons. Firstly, one of the major purposes of this paper is to validate the adequacy of application of GTCFC in COTS processors on pico-satellites in their real space missions. Secondly, such 8052-compatible processors have seen wide applications in small satellite projects.[4] Thirdly, as an important part of the product line of the authors' research institutes, the OBCs have successfully fulfilled all their engineering tasks and scientific goals in the pico-satellites' real space missions since launched in September 2010, and have their on-orbit life expectancies well extended. Thus, they are considered qualified as standard equipment on successive satellite products under development. Last but not least, the goals and missions of these satellites comprise the research of this paper.

Suppose gtG of the target program has no more than 255 (virtual) vertices, i.e., $\mathrm{card}(VV) \leqslant 255$. This makes the lengths of the assigned IDs no longer than the word length of the platform (8 bits). Thus the demand for the fast processing recommendation in Section 2.3 is fulfilled.

4.1. Detection latency

Detection latency can be quantified in processor cycles, and is defined as the number of processor cycles to run between the terminal of the illegal branching and the line detecting the error here. As pointed out in Section 3, the detection of single branching errors in GTCFC takes place at the checking line L3 of the terminal basic block of the error, and that of multiple ones at the same line of the runtime successor of the terminal basic block. Given that the terminal lines of branching errors can be considered to submit to the even probability distribution, the expectation of the detection latency of single branching errors ($\overline{\mathrm{DL\_s}}$) is the average of the average detection latency of such errors terminating at each basic block, or:

$$\overline{\mathrm{DL\_s}} = \sum_{i=1}^{N} \frac{L(i,1,3)}{2} \times \frac{L(i,1,3)}{\mathrm{LT}} \tag{1}$$

where $L(i, p, q)$ is the number of processor cycles to run from the pth to qth region of basic block i, and LT is the length of the target program.Similarly, that of multiple branching errors $\overline{DL\_m}$ is

$$\overline{\mathrm{DL\_m}} = \sum_{i=1}^{N} \left(\frac{L(i,3,6)}{2} + \frac{L(i,1,3)}{\mathrm{card}(V)} \times \frac{L(i,3,6)}{\mathrm{LT}} \right) \tag{2}$$

Thus, the overall average detection latency $\overline{\mathrm{DL}}$ is

$$\overline{\mathrm{DL}} = \overline{\mathrm{DL\_s}} \times \sum_{i=1}^{N} \frac{L(i,1,3)}{\mathrm{LT}} + \overline{\mathrm{DL\_m}} \times \sum_{i=1}^{N} \frac{L(i,3,6)}{\mathrm{LT}} \tag{3}$$

It can be reasonably observed that the expectancy of detection latencies is in positive correlation to the average length of basic blocks. Additionally, it can be easily validated that the sample average of the detection latencies is an unbiased estimation of $\overline{\mathrm{DL}}$, and converges to the latter as the sample volume of branching error detection escalates.

4.2. Memory overhead

Each of the variables pVID and cVID takes up a memory space of the length of the assigned IDs, or 1 byte under the assumption above. In the meantime, arrays cVVID and RIP take up memory spaces of $(\mathrm{card}(V) + 1)$ and $(\mathrm{card}(VV) + 1)$ times the assigned IDs respectively. Thus, the memory overhead of GTCFC is of a linear complexity.

4.3. Instruction and execution time overheads

Statistics show that it takes the target program 19 more instructions, 32 more bytes for instruction storage, and 41 more cycles of execution time per basic block to be hardened by GTCFC on the OBC. For the whole program, the overheads are those times $\mathrm{card}(V)$ respectively. Note that the inequivalence among the three figures is due to the CISC nature of 8052.

5. Test results and discussion

Tests of GTCFC for various target programs, including bubble sort, insertion sort, matrix multiplication, and binary search are carried out on the OBC. Despite that they are processor-specific, their results are mostly significant in a general sense for the same reason given in Section 4.

Four target programs are chosen for the tests for a certain series of reasons. Firstly, they present certain varieties of control flow graph patterns. Bubble and insertion sorts and binary search take a lot of branching among relatively simple calculations, leaving more substantial overheads. While matrix multiplication carries out a lot of time-consuming multiplying and much less branching. At the same time, these target programs make use of almost all instructions available for 8052, including those for arithmetic and logic calculations, and branching. Comparisons of the targets to the real program of the OBC for

its current space mission shows that the formers take similar control flow graph patterns and apply to merely the same instruction sets as the latter. Therefore, the formers are adequate representatives for the latter in such senses. Meanwhile, the target programs are merely the most common standardized algorithms that see a great deal of applications. This allows them being even more representative.

The tests are composed of four steps, namely benchmark generation, tests for detection capability, performance and overhead with and without injections, and comparison tests. Details of the steps are given below.

Step 1 An original target program executes, with a timer interrupt measuring its execution time as benchmark.

Step 2 The GTCFC-hardened target program executes with the same timer interrupt measuring its execution time for calculating the execution time overhead.

Step 3 The GTCFC-hardened target program executes with error injections. Two interrupt processes are introduced in this step. A timer interrupt is used similarly as in the previous two steps for measuring the execution time of the hardened program and detection latencies of errors, while a serial-port interrupt serves as a software runtime injector for branching errors. It does so by receiving injection commands from a host computer, and making correspondent jumps upon its exit to the designated addresses by the commands within the memory occupied by the hardened target program. Instead of formalizing the compositions and structures of the pools of the injected errors, they are generated in a randomized pattern to simulate real branching errors to the fullest extent possible. The occurrence time of the injections is randomly given by the host. That is to say, the original addresses of the branching errors are random. Meanwhile, the terminal addresses given by the injection commands are generated randomly. Furthermore, a sample volume of 10000 injections is obtained for each test to ensure their statistical credibility.

Step 4 Comparison tests of GTCFC versus two preeminent algorithms with well-established high performance and low time overheads among the enumerated previous work, ECCA and CCA,[9] are also carried out. Insertion sort and matrix multiplication are chosen as targets due to their radically different patterns of control flow graphs and lengths of basic blocks. For these tests, basic block designations, error handling, and benchmarking and testing schemes are all the same as those for GTCFC (and as in the previous three steps). Meanwhile, the same statistical parameters are extracted from the results.

The results of the first three steps of the tests are given in Table 1, in which an overhead rate means the ratio of the extra resource used by GTCFC compared to that used by the original target program before hardening. Note that although the injected errors are randomized despite their classifications, statistics of the results show that all five types of injected errors have been successfully detected by GTCFC for each target program. Additionally, the results are actually uncorrelated with the workloads of the target programs. This is because the figures of detection capability, performance and overhead are either irrelevant to or measured in proportion to them.

The three steps show that GTCFC can detect over 90% of all the randomly injected branching errors. There are several reasons for the minority of the injected errors not being detected. Firstly, since the errors can cause incomplete execution of the multiple-instruction checking lines, correspondent detection omissions are possible although highly unlikely. Additionally, injections can result in no branching errors, or incorrect conditional and intra-basic-block branching errors. The first case poses no errors, while the latter two are considered as data errors and not supposed to be detected by GTCFC as stated above. Yet none of these cases are excluded from the tests.

Memory, instruction, and execution time overheads are target-sensitive parameters. For the tested programs, approximately 20 bytes of memory are spent on storing the extra data structures. Instruction and execution time overheads vary due to the differences in control flow graph patterns and lengths of basic blocks. Due to the differences in control flow graph patterns of the target programs stated above, it takes significantly lower ratios of instructions and execution time for GTCFC to harden matrix multiplying than bubble and insertions sorts. The average of branching error detection latencies of different programs are in proportion to the average lengths of their basic blocks, as Section 4.1 suggests. Therefore, it is reasonable that it takes GTCFC a much longer average latency to detect branching errors in matrix multiplication than in others.

The results of the Step 4 (comparison tests) are shown in Table 2.

Observably from the statistics, it takes ECCA more time (instruction and time overheads, and detection latencies) to achieve lower error detection rates than GTCFC for both targets. This is mainly because the former employs multiplication and division for signature transfers. Such operations are time- and instruction-consuming on processors without multipliers like the one in question, and bring out higher error probabilities in executing the inserted instructions. CCA projects a slightly higher detection capability and less overhead for insertion sort than GTCFC. But for matrix multiplication, more overheads (although less than ECCA) are cost to yield a lower detection rate than for GTCFC. This reflects its lousier detection capability of certain categories of errors.[9] Yet for both target programs, GTCFC needs more memory space than the others as expected.

Such results suggest that GTCFC not only offers a high detection capability of errors and performs steadily well for different types of targets, but also does so with low latency and time overheads, and without having to rely on specific hardware. Meanwhile, its major requirement is only a memory space with complexity of $O(\mathrm{card}(V) + \mathrm{card}(VV))$ as explained earlier. On the contrary, although both ECCA and CCA take up roughly no extra memory spaces, the former resorts to hardware multipliers to realize its high performance and low time overheads, or decays to a slow and weak one when working without them, while the detection capability, and performance and time overheads of the latter are drastically variable and often worse than GTCFC.

The test results also show that GTCFC is adequate for application on COTS processors on pico-satellites in their real space missions. The current space missions of the ZDPS-1A pico-satellites take on a lot of similarities to those of other pico-satellites developed across the globe[23–27] in capabilities, workloads, and resource consumptions and margin levels of OBCs, orbital conditions and life expectancies. Therefore, the profile of its current space missions with significance in a general sense is chosen here as a grounding for the validation.

On one hand, GTCFC can be appended into the OBCs of the ZDPS-1A pico-satellites conveniently with still sufficient system margins left. The capability of the processor of the

Table 1 Results of the Steps 1–3 of the tests of GTCFC.

Target program	Injected error	Detected error	Detection rate (%)	
Bubble sort	10000	9057	90.57	
Insertion sort	10000	9092	90.92	
Matrix multiplication	10000	9425	94.25	
Binary search	10000	9042	90.42	
Target program	Memory overhead (byte)	Instruction overhead rate (%)	Average execution time overhead rate	Average detection latency (cycle)
Bubble sort	20	205.13	70.88	117.92
Insertion sort	26	222.56	60.73	95.38
Matrix multiplication	20	26.75	19.32	142.96
Binary search	20	30.37	60.21	116.26

OBC has been described in Section 4. It takes roughly 30% of its time to run the real program for the current space mission, waiting for tasks to come by running NOP instructions for the other 70% of time. Thus, even if the maximum execution time overhead tested of 71% is imposed, the processor still has an approx. 49% overall idle time. The real program uses approx. 1 out of the 2 kbytes data memory. The graph tree notation of its control flow graph shows that it has approx. 150 vertices and 250 virtual vertices. Thus, it can be calculated that the data memory can still bear a 30% spare space with GTCFC hardening the program. A space of 18 kbytes out of the 62 kbytes program memory is used by the real program. Thus, the worst-case instruction overhead tested will have the program space expense enlarged to approximate 60 kbytes. Although the margin left is not significant, it can be accepted since the onboard programs in pico-satellite space missions are usually firm. Even if they are not, the program memory of the OBC can be easily expanded off their processors. Moreover, since GTCFC imposes the need of neither utilizing any external hardware to the processors, nor speeding them up to compensate the execution time overhead, it does not increase the power consumption of the OBC, or ruin the overall energy balance of the pico-satellites.

On the other hand, GTCFC has the ability to minimize the risk of control flow errors being undetected to the space mission of the pico-satellites. The ZDPS-1A pico-satellites, like most others do, operate on low-earth orbits (with altitudes of approximate 650 km). The rate of SEU in RAMs on that altitude was measured on orbit to be $4.21 \times 10^6/(\text{bit} \times \text{day})$ by SAMPEX from Godard Space Flight Center of National Aeronautics and Space Administration (NASA).[28] Whilst it is reported that the rate for floating-gate ROMs is five orders of magnitude lower than that of the RAMs under the same radiation input.[29] Therefore, during the expected on-orbit life of 90 days of the ZDPS-1A pico-satellites, the 2 kbytes (16 kbits) data RAM of the OBC would experience a total of 6.0624 upsets, while that figure of the program ROM is a negligible 1.88×10^3. Since GTCFC can detect at least 90% of all control flow errors, even if all of the upsets are prone to cause such errors, GTCFC can reduce the total number of them to 0.6 during the 90 days. This means that the GTCFC-hardened OBC can operate mostly safely with at most one control flow error unattended throughout the on-orbit life of its host pico-satellite.

Nevertheless, there are certain limitations within the GTCFC solution. GTCFC requires a memory space of a linear complexity as has been mentioned. In cases of the target programs having extremely abundant branching, such an overhead will be more of a significant drawback compared to other software solutions with fixed memory overheads. Additionally, GTCFC cannot handle branching errors that jump to unused program memory. However, in fact, such errors are not in the domain of the software control flow checking solutions, thus not meant to be handled by them, including the ones presented in the Introduction Section. Instead, in space missions, a technique called interrupt trapping is usually

Table 2 Results of comparison tests of GTCFC vs ECCA and CCA.

Target program	Hardening algorithm	Injected error	Detected error	Detection rate (%)	
Insertion sort	GTCFC	10000	9092	90.92	
	ECCA	10000	7928	79.28	
	CCA	10000	9134	91.34	
Matrix multiplication	GTCFC	10000	9425	94.25	
	ECCA	10000	7760	77.60	
	CCA	10000	7559	75.59	
Target program	Hardening algorithm	Memory overhead (byte)	Instruction overhead rate (%)	Average execution time overhead rate (%)	Average detection latency (cycle)
Insertion sort	GTCFC	26	222.56	60.73	95.38
	ECCA	1	595.49	319.02	196.70
	CCA	3	218.80	57.10	78.37
Matrix multiplication	GTCFC	20	26.75	19.32	142.96
	ECCA	1	40.86	54.50	763.16
	CCA	3	39.23	26.94	422.38

applied to tackle these problems. The technique fills all the unused program memory with an unconditional branching instruction to the entrance of the target program. Thus, whenever a branching error lands in the unused memory, the processor is reset by such an instruction.

6. Conclusions

This paper proposes a generic high-performance and low-time-overhead software control flow checking solution, GTCFC. Firstly, a tree notation for graphs is introduced, and control flow checking is made possible by making use of the tree characteristic that each vertex has only one parent. Then the algorithm is proven to bear the capability of detecting all single and multiple errors while remaining generic, or applicable in arbitrary instruction sets and independent of any specific hardware. Additionally, a detailed analysis reveals its low detection latency and time overheads, and linear-complexity memory overhead. In the test phase, GTCFC not only performs in consistency with the theory to detect over 90% of the randomly injected and all-pattern-covering branching errors with low detection latency and time overheads, but beats two well-established preeminent control flow checking algorithms, CCA and ECCA, in these dimensions. The results are further analyzed to reach such a conclusion that GTCFC not only can be accommodated in pico-satellites conveniently with still sufficient system margins left, but also has the ability to minimize the risk of control flow errors being undetected in their space missions. At the same time, limitations of the solutions are also pointed out. Although the analysis and tests are instantiated on COTS-processor-based OBC of in-service ZDPS-1A pico-satellite products, they logically stay generally significant. Therefore, due to its effectiveness, efficiency, and compatibility, GTCFC is theoretically proven and experimentally validated ready for applications on COTS processors on pico-satellites in their real space missions. Future work mainly lies in mitigating its limitations, and experimenting and applying it in proposed space missions of ZDPS-1A pico-satellites and their successors.

Acknowledgement

This work was supported by National Natural Science Foundation of China (No. 60904090).

References

1. Wallmark JT, Marcus SM. Minimum size and maximum packaging density of non-redundant semiconductor devices. In: *Proceedings of the IRE*, 1962. p. 286–98.
2. Baumann RC. Radiation-induced soft errors in advanced semiconductor technologies. *IEEE Trans Device Mater Rel* 2005;**5**(3):305–16.
3. Bhattacharya K, Ranganathan N. RADJAM: a novel approach for reduction of soft errors in logic circuits. In: *22nd international conference on VLSI design*; 2009. p. 453–8.
4. Heidt H, Puig-Suari J, Moore AS, Nakasuka S, Twiggs RJ. CubeSat: a new generation of picosatellite for education and industry low-cost space experimentation. In: *Proceedings of the 14th Annual AIAA/USU conference on small satellites*; 2000. p. 113–6.
5. Rajabzadeh A, Miremadi SG. A hardware approach to concurrent error detection capability enhancement in COTS processors. In: *11th Pacific Rim international symposium on dependable, computing*; 2005. p. 8.
6. Lai HC, Horng SJ, Chen YY. An online control flow check for VLIW processor. In: *14th IEEE Pacific Rim international symposium on dependable, computing*; 2008. p. 256–64.
7. Michel T, Leveugle R, Gaume F, Roane R. An application specific microprocessor with two-level built-in control flow checking capabilities. In: *Proceedings of Euro ASIC' 92*; 1992. p. 310–3.
8. Oh N, Shirvani PP, McCluskey EJ. Control-flow checking by software signatures. *IEEE Trans Rel* 2002;**51**(2):111–22.
9. Alkhalifa Z, Nair VSS, Krishnamurthy N, Abraham JA. Design and evaluation of system-level checks for on-line control flow error detection. *IEEE Trans Parall Distr Syst* 1999;**10**(6):627–41.
10. Lu DJ. Watchdog processor and structural integrity checking. *IEEE Trans Comput* 1982;**C-31**(7):681–5.
11. Miremadi G, Harlsson J, Gunneflo U, Torin J. Two software techniques for on-line error detection. In: *22nd international symposium on fault-tolerant, computing*; 1992, p. 328–35.
12. Furtado P, Madeira H. Fault injection evaluation of assigned signatures in risc processors. In: *2nd Europe conference on dependable, computing*; 1996. p. 55–72.
13. Wilken K, Shen JP. Continuous signature monitoring: low-cost concurrent-detection of processor control errors. *IEEE Trans Comput Aid D* 1990;**9**(6):629–41.
14. Madeira H, Silva JG. On-line signature learning and checking: experimental evaluation. In: *5th annual European computer conference on advanced computer technology, reliable systems and applications*; 1991. p. 642–6.
15. Borin E, Wang C, Wu Y, Araujo G. Software-based transparent and comprehensive control-flow error detection. In: *Proceedings of the international symposium on code generation and, optimization*; 2006. p. 333–45.
16. Huang J, Li Y, Zhang L, Xie Y, Han C. A novel high-capability control-flow checking technique for RISC architectures. In: *International conference on embedded software and systems*; 2008. p. 258–63.
17. Yau SS, Chen FC. An approach to concurrent control flow checking. *IEEE T Software Eng* 1980;**2**:126–37.
18. Benso A, Di Carlo S, Di Natale G, Prinetto P, Tagliaferri L. Control-flow checking via regular expressions. In: *Proceedings of 10th Asian test, symposium*; 2001. p. 299–03.
19. Michel T, Leveugle R, Saucier G. A new approach to control flow checking without program modification. In: *21st international symposium on fault-tolerant, computing*, 1991. p. 334–41.
20. Motohashi S, Nakasuka S, Aoki T, Narusawa Y, Nagashima R, Kawakatsu Y, et al. On-orbit dynamics and new control scheme for large membrane Furoshiki Satellite. In: *21st international symposium on space technology and, science*; 1998. p. 1072–7.
21. Schaffner JA, Puig-Suari J. The electronic system design, analysis, integration, and construction of the Cal Poly State University CP1 cubeSat. In: *16th AIAA/USU conference on small satellites*, 2002.
22. Zhang Y, Yu FX, Zheng YM, Huang ZL, Chen L, Jin ZH. Fault tolerance design of pico-satellite's house keeping system. *J Astronaut* 2007;**28**(6):1753–7 [Chinese].
23. Toorian A, Diaz K, Lee S. The cubeSat approach to space access. In: *IEEE aerospace conference*, 2008. p. 1–14.
24. Motohashi S, Nakasuka S, Aoki T, Narusawa Y, Nagashima R, Kawakatsu Y, et al. On-orbit dynamics and new control scheme for large membrane Furoshiki satellite. In: *21st international symposium on space technology and, Science*; 1998. p. 1072–7.
25. Narusawa Y, Aoki T, Nakasuka S, Motohashi S, Nagashima R, Kawakatsu Y, et al. Behavior of membrane structure under microgravity environment. In: *Proceedings of 21st ISTS*; 1998. p. 428–33.
26. Motohashi S, Nakasuka S. On-orbit dynamics and control of large scaled membrane with satellites at its corners. In: *Proceedings of IFAC symposium on aerospace control*; 1998. p. 146–51.

27. Chang YK, Park JH, Kim YH, Moon BY, Min MI. Design and development of HAUSAT-1 picosatellite system (cubesat). In: *International conference on recent advances in space technologies*; 2003. p. 47–54.
28. Seidleck CM, LaBel KA, Moran AK, Gates MM, Barth JM, Stassinopoulos EG, et al. Single event effect flight data analysis of multiple NASA spacecraft and experiments: implications to spacecraft electrical designs. In: *Third European conference on radiation and its effects on components and systems*; 1995. p. 581–8.
29. He CH, Geng B, Yang HL, Chen XH, Li GZ, Wang YP. Comparison and analysis or radiation effects between floating gate ROMs and SRAMs. *Acta Electron Sin* 2003;**31**(8):1260–2 [Chinese].

Yang Mu received his B.Eng. degree in information science and electronic engineering at Zhejiang University. He is currently a doctoral candidate in Department of Information Science and Electronic Engineering at Zhejiang University. His main research interests are onboard computers and composite electronic technologies for micro-satellites.

Wang Hao received his B.S. degree in industry automation in 1995, and M.S. (2002) and Ph.D. (2007) degrees in instrumentation from Nanjing University of Aeronautics and Astronautics and Shanghai Jiao Tong University respectively. Since 2007, he has worked in the Department of Information Science and Electronic Engineering at Zhejiang University. He is currently an associate professor and his major technical activities have been in attitude control system design for micro-satellites.