DOI: 10.2298/CSIS120218056T

Formal Verification of Signature-monitoring Mechanisms by Model Checking

Lanfang Tan, Qingping Tan, Jianjun Xu, and Huiping Zhou

School Computer, National University of Defense Technology,
410073 Changsha, China
{tanlanfang1022, eric.tan.6508, jjun.xu, icent}@gmail.com

Abstract. In recent decades, reliability in the presence of transient faults has been a significant problem. To mitigate the effects of transient faults, fault-tolerant techniques are proposed. However, validating the effectiveness of fault-tolerant techniques is another problem. In this paper, we present an original approach to evaluate the effectiveness of signature-monitoring mechanisms. The approach is based on model-checking principles. First, the fault tolerant model is proposed using step-operational semantics. Second, the fault model is refined into a state transition system that is translated into the input program of the symbolic model checker NuSMV. Using NuSMV, two reprehensive signature-monitoring algorithms are verified. The approach avoids the state space explosion problem and the verification was completed with practical time. The verification results reveal some undetected errors, which have not been previously observed.

Keywords: software fault-tolerance, model checking, formal verification, fault tolerance, signature monitoring mechanisms.

1. Introduction

In recent years, with the growing demand for high availability and reliability of computer systems, validating the effectiveness of fault-tolerant techniques constitutes a significant problem [1]. Fault injection techniques have been instinctively proposed, in which faults are injected into target systems to see whether the fault tolerant procedures could properly behave [2]. However, fault injection techniques have a common drawback, which is their failure to cover all fault scenarios. These techniques also tend to be very time consuming. Therefore, the development of efficient verification techniques for fault-tolerant systems is urgent. Given that the formal verification technique can target all possible "corner cases," which may be missed by conventional fault injection methods, the verification technique for fault-tolerant systems becomes a promising candidate solution. A few methods for formal verification of fault-tolerant techniques have been proposed. Generally, these methods can be classified into two categories:

- Deductive verification: a technique that proves the correctness of fault-tolerant systems using axioms and proof rules [3]. The intrinsic advantage of the deductive technique is its ability to reason out all nondeterministic errors. However, it can only be used by experts who are educated in logical reasoning and have considerable experience. Deductive verification also cannot be automatically performed.
- Model checking: In earlier research [4][5][16-18], model checking was proposed to evaluate the reliability of fault-tolerant systems. It explores all possible system states to cover all fault scenarios. However, it may suffer from the state space explosion problem.

Though model-checking techniques are promising, many theoretical questions remain. In particular, a general approach based on model checking for the evaluation of fault-tolerant techniques has not been formulated. An approach is usually specialized for a specific system, and different techniques need different validation paradigms. For example, Nicolescu et al. verified a control-flow checking technique by constructing a hypothetical program that augmented the technique, and then checks the program for undetected errors [4]. For other detection mechanisms, construction of similar programs is difficult.

In this paper, we propose a general approach to evaluate the effectiveness of signature-monitoring techniques [9-11] using model checking. It aims at formally proving the capacity of the signature-monitoring techniques to detect errors over a general class of possible applications. In other words, can we verify that some control-flow errors (CFEs) can be detected by the signature-monitoring technique, while some CFEs cannot? The purpose is to catch all undetected CFEs that potentially escape from conventional detection. The main contributions of our work are highlighted as follows:

- An original approach for the evaluation of signature-monitoring techniques is proposed, which provides a universal validation paradigm for all signature-monitoring mechanisms.
- A fault tolerant model is proposed to describe the execution of the assembly program that is strengthened by the signature-monitoring mechanism. A state transition system is refined and a procedure on how to translate the state transition system into the input program of the model checker NuSMV is explained in detail. The translation procedure can be applied to all cases.
- Two reprehensive signature-monitoring algorithms are evaluated. Some undetected errors that have not been found before are revealed. Especially for the dynamic signature-monitoring (DSM) technique, which can detect all illegal interblock errors [4][9], four kinds of errors that escaped detection were found.

The remainder of this article is organized as follows. Section 2 lists the related work . Section 3 describes the basic principle of the signature-monitoring mechanism. Section 4 presents the fault tolerant model for the assembly programs that are strengthened by the signature-monitoring mechanism. Section 5 describes the state transition system refined by the fault tolerant model and the translation of the state transition system into the input language of the model checker. In Section 6, two representative algorithms are verified and the verification results are analyzed. Section 7 lists the comparisons. Section 8 states the conclusions and future works.

2. Related Work

Some earlier formal verification techniques have been proposed to evaluate the system dependability. Nicolescu et al. [4] proposed an original approach to evaluate the system reliability with respect to transient errors. Based on a generic model over a general class of all possible applications, the proposed validation approach allows exploring all fault scenarios. The validation confirms that DSM technique satisfies the property that if an error corrupts the control flow execution, it will ultimately be detected. Our approach verifies the DSM technique and the comparisons are illustrated in Section 7.

Tomoyuki et al. [5] proposed a symbolic model checking method for verification of fault tolerance of systems. At first the fault tolerant system is specified in the form of a guarded-command program. A modeling language suited for describing guarded-command programs is defined, and then a translation method from the modeling language to the SMV language is proposed. By specifying the fault tolerance property as a CTL formula, the examples were verified to demonstrate the usefulness of the proposed method. The difference with our work is the target fault model. The faults in [5] mainly are crash faults and byzantine faults, while our study focuses on transient faults [19].

Arora and Gouda [16] first gave a formal definition of “fault tolerance” for a system, which consists of a safety requirement, closure and convergence. Then a formal framework for reasoning about fault-tolerant system was proposed, which enables reasoning independent of technology, architecture and application considerations. Similar to the work in [5], this method deals with the stuck-at, crash, failstop, omission and byzantine faults.

John [17] presented a methodology that can ease the formal specification and assurance of critical fault-tolerant system. The functional program is first transformed into an untimed synchronous system and then into its time-triggered implementation. The first step is specific to the algorithm concerned, but the second is generic and its correctness was proved. This proof was formalized and mechanically checked with the PVS verification system. This work is applicable to the critical real-time applications.

Daniel and Ruben [18] proposed a method of first using aspect oriented programming (AOP) to add fault tolerance to software, and then formally verifying the fault tolerance property of a program. Meng et al. [20] analyzed the impact on formal verification effort and testing effort due to adding different fault tolerance mechanisms to baseline systems. By comparing the experimental results of different designs, they concluded that re-execution (time redundancy) was the most efficient mechanism, followed by parity code, dual modular redundancy (DMR), and triple modular redundancy (TMR). Moreover, the ratio of verification effort to testing effort to assist designers in their tradeoff analysis when deciding how to allocate their budget between formal verification and testing was defined, which could be used in practical industrial production.

3. Signature-monitoring Mechanisms

In this paper, the target errors to be tolerated are transient errors [19], which emerge from external events such as energetic particles striking the chip due to the shrinking chip size and increasing transistor density of the modern processors. Transient faults do not cause permanent damage, but may result in incorrect program executions by altering signal transfers or stored values. In terms of effects, transient errors can be classified into data errors and CFEs. Data errors appear when the contents of variables in the memory or in the microprocessor registers are altered. CFEs refer to deviations from the program's normal instruction execution flow, such as upsets to the target addresses of branch instructions or the program counter (PC). Studies have shown that CFEs account for 33% to 77% of all transient errors [7]. Therefore, computer systems must be equipped with CFE detection mechanisms.

A multitude of CFE detection techniques have been proposed [8–11]. Signature-monitoring techniques are apparently the most universal and effective techniques. Several signature-monitoring techniques have been developed [9–11]. Control-flow checking by Software Signatures (CFCSS) [11] and DSM [9] are two representative signature-monitoring techniques, which will be verified in Section 6. As a general characteristic, signature-monitoring techniques are designed to detect illegal interblock transitions that are incorrect jumps to other blocks corrupting the program control flow. Signature-monitoring techniques are based on the partition of the program code into basic blocks (BB) [12]. A basic block is a maximal set of sequential instructions that start from the first instruction and terminate at the last instruction. No branch instruction exists in a basic block except for the last instruction.

After dividing the program into BBs, a unique static signature is associated to each BB during the pre-compilation phase. Some assistant signature variables are also introduced. Based on the signatures, checking instructions are added. When a program is executed, a dynamic signature variable is computed and compared with the static signature of the current executing block. If the dynamic signature equals to the static signature, no error occurs. Otherwise, the checking instructions see the mismatch and detect the error.

Checking instructions can be classified into three kinds in terms of their effects. Generation instructions produce the dynamic signature for the current block. Compare instructions identify whether the control has correctly reached this block. Preparing instructions update signatures for the transfer of control to the next block. For different signature-monitoring mechanisms, the locations and numbers of checking instructions are different. Fig. 1a shows an abstract description of how checking instructions are added. For some algorithms, preparing instructions may follow the original instructions. Whether preparing instructions emerge before or after the original instructions, the branch instruction should come last.

Fig. 1b illustrates how checking instructions are added in CFCSS. G and S represent the dynamic signature and static signature, respectively. D and d represent the assistant signatures. Two instructions “xor G,G,d_i” and “xor G,G,D” [11] update the dynamic signature G. The compare instruction “bne G, S_i, error” compares the dynamic signature with the static signature. If they mismatch, program control flow transfers to the error handler “error”. The instruction “xor D,S_h,S_i” generates the assistant signature D to prepare for the control flow transfer.

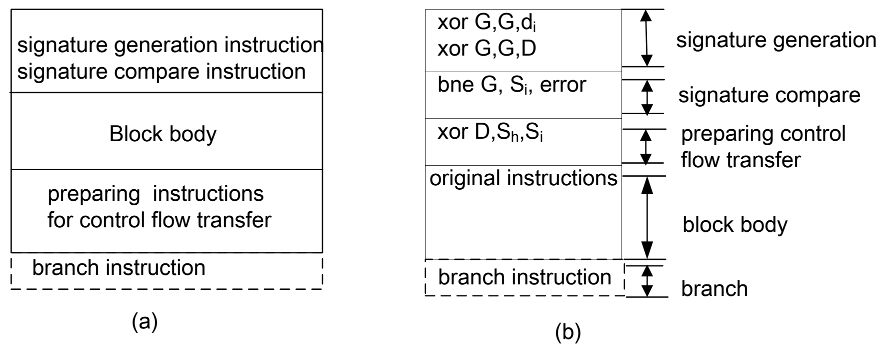


Fig. 1. (a) An abstract description of adding checking instruction in a basic block; (b) Example of checking instructions in CFCSS algorithm

4. The Fault Tolerant Model

A control flow machine is introduced to describe the control flow transfer of the fault tolerant program strengthened by signature-monitoring mechanism. Suppose the hardware affected by transient faults is based on a simple RISC architecture, the execution of the fault tolerant program strengthened by signature-monitoring mechanism can be specified using a step-operational semantics, which maps a machine state to other machine states.

4.1. The Syntax of the Machine

Considering that signature-monitoring mechanisms only focus on CFE, operations on the data segment (general-purpose registers and memory) can be abstracted from the machine. For clarity, we adopt a minimal assembly instruction set that is composed of control flow instructions jump (jmp) and conditional branch (brz), as well as signature-generation (generate), signature-compare (compare), and signature-preparing (prepare) instructions. Fig. 2 details the syntax of the machine states.

For clarity and elegance, assume that the values of signature and address are integers. Meta-variable *n* generally ranges over integers. When we emphasize that an integer is used as an address, the meta-variable *l* is used. Similarly, a meta-variable *s* is introduced to express the values of signature variables. *rz* denotes the branch condition of the conditional branch instructions, with the value “true” and “false”. *rt* denotes the target address of the control flow instructions.

Control flow instructions consist of jump instruction (jmp *rt*) and conditional branch instructions (brz *rz*, *rt*). Checking instructions consist of signature-generation (generate), signature-compare (compare), and signature-preparing (prepare) instructions. For different signature-monitoring techniques, the operations of checking instructions are different.

Instructions are grouped together in basic blocks BB. These blocks always begin with signature-generation and signature-compare instructions, are then filled with the block body and terminate by signature-preparing instructions. If a block has control flow transfer, control flow instruction (jmp *rt*) or (brz *rz*, *rt*) ends with the block. Since the original instructions of the block do not alter the program control flow, they can be abstracted as an entirety, in other words, the block body.

The machine states can be modeled as a tuple (*C*, *H*, *BB*, *S*, *PC*) that are composed of code memory *C*, history *H*, basic block being executed *BB*, signature variables *S*, and program counter *PC*. These are defined as follows:

– Code memory *C*$[l \to BB]$ is a partial map from addresses to valid basic blocks. Block addresses are all ordered. We use the notation *l*+1 to refer to the address of the block, which follows the block at *l*. If a block at *l* ends with a conditional branch, we assume that *l*+1 inhabit the domain of *C.* In other words, conditional branch always have a block to fall through to.
– History *H* is a sequence of labels that record basic blocks being executed during the current execution. When a program is completed executing, *H* denotes the execution path.
– Block *BB* denotes the basic block being executed.
– Signature variables *S* have two components, namely, the dynamic signature variable *G* and the assistant signature file *A*. *A*$[a \to s]$ is a mapping from assistant signature variables to the values they contain. Different signature-monitoring mechanisms introduce different assistant signature variables.
– Program Counter *PC* refers to the next instruction to be executed.

To describe the fault-tolerant program behavior after an error occurs, we introduce three special “final states”. The *detected* state represents a state in which an error has been detected, whereas the *invalid* state represents a state when an error causes transition to an invalid address. The *exit* state represents a state that program exits normally, though an error occurs.

Address values	l	::= n
Program Conter	PC	::= l
Branch condition	rz	::= true | false
Branch intention register	rt	::= l
Signature values	s	::= n
Dynamic signature	G	::= s
Assistant signature variables	a	::= a_1 |...a_n
Assistant signature file	A	::=. | A, a -> s
Signature variables	S	::= (G, A)
History	H	::=$l1$,...ln
Control flow instruction	i	::= jmp rt | brz rz,rt
Checking instruction	c	::= | generate G, a, s | compare G, s | prepare a, a, s
Blocks Body	b	::= original instruction list
Basic blocks	BB	::= generate G, a, s; compare G, s; b ; prepare a, a, s ; i | .
Code memory	C	::= . | C, l->BB
State	Σ	::= (C, H, BB, S, PC)
Final state	Γ	::= *detected* | *exit* | *invalid*

Fig. 2. Syntax of instructions and machine states

4.2. The Semantics of the Machine

This section formalizes the operational semantics of the control flow machine. Generally speaking, the operational semantics are defined as the rule (1), which expresses that state Σ_1 transfers to state Σ_2 by a step execution under certain conditions.

$$\frac{Conditions}{\Sigma_1 \xrightarrow{step} \Sigma_2} \qquad (1)$$

Rule (2) illustrates the fault model. As an established standard, the current study adhered to the Single Event Upset (SEU) model [7], which states that only one fault occurs per execution. With signature-monitoring mechanisms only focusing on CFE, SEU can be modeled as illegal transitions to other instructions that are different from the expected target. Under the semantics of the control flow machine, error occurrence can be described by the rule error, which expresses that when CFE occurs, the program control flow transfers to a random instruction rather than to the expected instruction (denoted as *ex-*

pect(*PC*)). Sometimes, a CFE may cause a program to jump to an invalid address. CFE may occur any time as the rule can apply any time.

$$\frac{PC' \neq (\exp ect(PC)) \ \&\& \ BB' \neq BB}{(C,H,BB,S,PC) \xrightarrow{error} (C,(H;BB),BB',S,PC')\ /\ invalid(H)} \quad (2)$$

Several notations are introduced to explain the following rules. Notation $S_{val}(a)$ denotes the value of *a*. Similarly, $R_{val}(rt)$ refers to the value of *rt*. Notation $S[a \mapsto s]$ corresponds to updated assistant signature variables file with variable *a* mapped to *s*. *PC*++ denotes incrementing *PC* by 1, when the execution of an instruction is complete.

Rule (3) expresses the execution of a jump instruction. The control flow is transferred to a new block when *rt* contains the address of a valid block.

$$\frac{R_{val}(rt) \in Dom(C) \ \&\& \ BB' = C(R_{val}(rt))}{(C,H,BB,S,PC) \xrightarrow{jmp\ rt} (C,(H;BB),BB',S,R_{val}(rt))} \quad (3)$$

The rule (4), which is for the conditional branch instruction, similarly follows jump instructions when the branch condition is satisfied.

$$\frac{R_{val}(rt) \in Dom(C) \ \&\& \ BB' = C(R_{val}(rt)) \ \&\& \ R_{val}(rz) = false}{(C,H,BB,S,PC) \xrightarrow{brz\ rz,rt} (C,(H;BB),BB',S,R_{val}(rt))} \quad (4)$$

Accordingly, the rule (5) describes that the program control flow proceeds to the next block when the condition is not satisfied.

$$\frac{R_{val}(rt) \in Dom(C) \ \&\& \ R_{val}(rz) = true}{(C,H,BB,S,PC) \xrightarrow{brz\ rz,rt} (C,(H;BB),C(BB)+1,S,PC++)} \quad (5)$$

The following rules describe state transitions by updating signature variables. The rule (6) indicates the generation of the dynamic signature of the current block, where the function *f* =*generate* (s_1, s_2) defines the operations for signature generation.

$$\frac{G' = generate(S_{val}(a),s) \ \&\& \ S' = (G',A)}{(C,H,BB,S,PC) \xrightarrow{generate\ G,a,s} (C,H,BB,S',PC++)} \quad (6)$$

The rule (7) describes the mismatch that the dynamic signature is not equal to the static signature. Hence, an error is detected and the program transfers to a final *detected*(*H*) state, where *H* represents the sequence of the blocks during execution.

$$\frac{G \neq s}{(C,H,BB,S,PC) \xrightarrow{compare\ G,s} \det ected(H)} \quad (7)$$

By contrast, the rule (8) describes when the dynamic signature equals the static signature of the current block, no error is detected and the program continues with its execution.

$$\frac{G = s}{(C,H,BB,S,PC)\xrightarrow{compare\ G,s}(C,H,BB,S,PC++)} \qquad (8)$$

The rule (9) indicates that assistant signature variables are prepared for transfer control to the next block, where the function *f* = *prepare* (s_1,s_2) defines the operations for updating the assistant signatures.

$$\frac{A' = prepare(a \mapsto S_{val}(a), s)\ \&\&\ S' = (G, A')}{(C,H,BB,S,PC)\xrightarrow{prepare\ a,a,s}(C,H,BB,S',PC++)} \qquad (9)$$

5. Implementation

In this section, we first refine the execution of the control flow machine into a state transition system, and then describe its translation to the input language of the model checker for automatic verification.

5.1. The State Transition System

As aforementioned, a state of the control flow machine is formally represented by a tuple Σ=(*C*, *H*, *BB*, *S*, *PC*). Transitions between these states formalize the effect of the execution of an instruction. The fault-tolerant program execution then can be refined into a state transition system with one assembly instruction per state.

To explain the component *C* in the transition system, all basic blocks are numbered and each one is uniquely identified by an address represented by a special symbol. The addresses of the instructions in the basic blocks are also represented by symbols. For instance, the address of the first basic block (BB_1) is "#BB_1". The addresses of the instructions in BB_1 are identified as #BB_1(gen), #BB_1(comp), #BB_1(body), #BB_1(pre), and #BB_1(bran).

In the machine state, *H* represents the sequence of blocks involved during the current execution. When the execution is complete, *H* becomes the execution path. In an assembly program, if the branch conditions are identified, their execution paths are also identified. To express all execution paths, a tuple *Con*=(c_1,c_2,…c_N) is introduced, where c_i is a Boolean variable that denotes the branch condition of BB_i and N is the total number of basic blocks in the program. If BB_i is not a branch block that exits with a branch instruction, c_i is reserved as "false" and not modified by the model-checking tool during the verification.

The states of control flow machine can be simplified by a tuple *Ω* = (*PC*, *S*, *Con*, *error*), where *Ω* is the set of states of the state transition system. The state variables are defined as follows:

- *PC*: a Program Counter variable *PC* explicitly controls the sequencing of instructions. As aforementioned, the addresses of all basic blocks are represented by symbols. Therefore, the domain of *PC* can be represented by a set {#BB_1(gen), #BB_1(comp), #BB_1(body), #BB_1(pre), #BB_1(bran), ………, #BB_N(gen), #BB_N(comp), #BB_N(body), #BB_N(pre), #BB_N(bran), Normal_exit, Fault_detected and Invalid_address}. To describe the final states, three special symbols, namely, "Normal_exit", "Fault_detected" and "Invalid_address" are introduced. In the absence of error, program normally exits and *PC* eventually transforms to "Normal_exit". Conversely, if an error occurs and is caught by the signature-monitoring mechanism, *PC* transforms to "Fault_detected". Moreover, if an error causes transition to an invalid address, *PC* transforms to "Invalid_address".
- *S* is a set of signature variables {G, a_1,a_2,..a_M} that are introduced to represent the signature information, where *G* denotes the dynamic signature, a_i denotes the assistant signature variable and *M* denotes the number of introduced assistant signature variables. The values of the signature variables are updated upon executions of signature generation and preparing instructions. In the CFCSS algorithm, for example, when the generation instruction "xor *G*, *G*, d_i" is executed, the dynamic signature *G* is updated as "$G \oplus d_i$". Similarly, when the preparing instruction "xor *D*, S_h, S_k" is executed, the assistant signature *D* is updated as "$S_h \oplus S_k$". The symbol "$\oplus$"denotes the bitwise XOR operation. Different signature-monitoring mechanisms have different signature computations.
- *Con*: To express all execution paths, a tuple *Con*=(c_1,c_2,…c_N) is introduced. During verification, the values of (c_1,c_2,…c_N) are comprehensively modified by the model-checking tool, thus allowing the exploration of all execution paths.
- *error* is introduced to specify when a CFE occurs. According to the fault model, an error can occur any time during the execution. To describe the occurrence of the error, a set of Boolean expressions {*error* == #BB_1(gen), *error* == #BB_1(comp), *error* == #BB_1(body), …, *error* == #BB_N(body), *error* == #BB_N(pre) and *error* == #BB_N(bran) } are introduced. Only one expression can be true in an execution and the identification of the true expression is random. For instance, if the value of "*error* == #BB_i(body)" is true, it denotes that an illegal transition is transferred from the original instructions of BB_i. And the destination of the illegal transition is comprehensively modified by the model-checking tool, thus covering all fault scenarios.

The transition system can then be modeled as $< \Omega, A, \rightarrow, \Theta >$, where Ω is the set of states, A denotes the set of actions, $\rightarrow$ denotes the transition relation, and Θ is the set of initial states. A state is initial if and only if *PC* is equal to the first instruction of the entry block, and then an initial value is declared for every other state variable. For signature variables, their initial values are determined by the signature-monitoring mechanism. For variables in *Con* and *error*, their initial value is randomly selected in their pre-defined domain. By modeling each instruction execution as an action, the transition is defined as $\Omega_1 \rightarrow \Omega_2$, where Ω_1 and Ω_2 are the current and next states.

Fig. 3 demonstrates an example how states transfer with the execution of the program. Fig. 3a illustrates the execution of a basic block, where the solid lines show the correct control flows and the dashed line represents the illegal transition caused by a CFE. The corresponding state transitions of Fig. 3a are shown in Fig. 3b. The circles denote the states, which are represented by a tuple that contains the assignment of current values to the state variables. By modeling each instruction execution as an action, the current state is transferred to the next state with some variables updating by the execution. If an error occurs, program control flow transfers to a random instruction that is different from the expected one. Correspondingly, the current state transfers to the next state that is not the expected one, such as the transition from Ω_3 to Ω_r that illustrates the occurrence of a CFE.

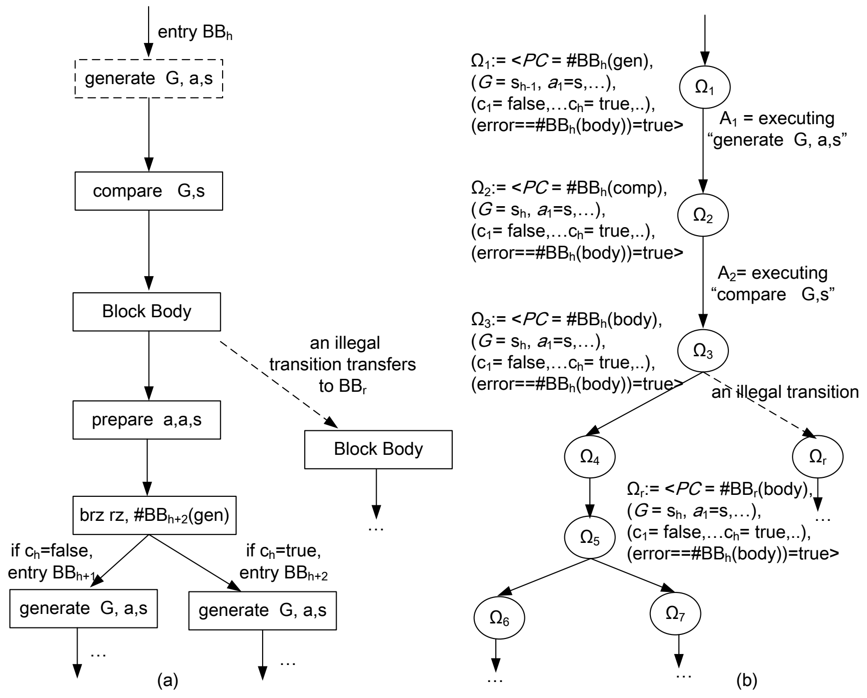

entry BB_h
generate G, a,s
compare G,s
Block Body
an illegal transition transfers to BB_r
Block Body
prepare a,a,s
brz rz, #BB_h+2(gen)
if c_h=false, entry BB_h+1
if c_h=true, entry BB_h+2
generate G, a,s
generate G, a,s
(a)
Ω1:= <PC = #BB_h(gen), (G = s_h-1, a1=s,…), (c1= false,…c_h= true,..), (error==#BB_h(body))=true>
A1 = executing "generate G, a,s"
Ω2:= <PC = #BB_h(comp), (G = s_h, a1=s,…), (c1= false,…c_h= true,..), (error==#BB_h(body))=true>
A2= executing "compare G,s"
Ω3:= <PC = #BB_h(body), (G = s_h, a1=s,…), (c1= false,…c_h= true,..), (error==#BB_h(body))=true>
an illegal transition
Ωr:= <PC = #BB_r(body), (G = s_h, a1=s,…), (c1= false,…c_h= true,..), (error==#BB_h(body))=true>
Ω1
Ω2
Ω3
Ω4
Ω5
Ω6
Ω7
Ωr
(b)

Fig. 3. Examples of state transitions

5.2. Translation to NuSMV

In the current study, the model-checking technique is adopted to verify the effectiveness of signature-monitoring mechanisms. The model checker NuSMV is selected for two reasons. First, the input language of NuSMV is syntactically and semantically similar to the general description of a finite state transition system. Second, NuSMV has a powerful symbolic representation and an expressive Computation Tree Logic (CTL) that specify properties to be checked. To verify a finite system in NuSMV, it has to be described in the SMV input program. The translation of the state transition system into an SMV program is then described.

An SMV program is composed of variable declaration, state initialization, state transitions and a list of properties written in CTL formula. The complete syntax of the SMV language is described in the SMV documentation [6]. The steps for the translation are as follows:

Step1: Defining an SMV module

The state transition system can be described using a model. The SMV program must have a main MODULE, so the system can be represented by the main MODULE.

Step2: Defining each state variable

Each state variable in the transition system is directly mapped into an SMV variable. An SMV variable for each variable in the transition system is then declared and its domain is pre-defined. The keyword VAR is used to declare variables.

Step3: Defining the initial state

Each SMV variable is assigned to its initial value, using the INIT statement.

Step4: Defining transition relations

First, for each transition in the state transition system, the current state is defined with its enabling conditions in the DEFINE statement. For a transition from Ω_1 to Ω_2 , as shown in Fig. 4, its corresponding DEFINE statement is "t_1:=(PC = #BB$_h$(gen))∧(G=S$_{h-1}$∧a=s∧…) ∧(c_1=false∧…∧c_h=true ⋯)∧(error==#BB$_h$(body)) = true⋯)" .

Second, for each SMV variable v, a next statement is declared, which is expressed as follows:

```
next(v):=case
        t1 : v′;
        t2 : v′′;
        ......
        TRUE:v;
        esac;
```

Thus, if condition t_1 is true, v is updated as v'. Similarly, v is updated as v'' when t_2 is satisfied. Otherwise, no condition is true, v keeps the original value.

Step5: Defining the constraints for the error model

According to the SEU model, only one fault occurs during an execution. Hence, a constraint should be declared under the keyword INVAR. As aforementioned, a set of Boolean expressions {*error* == #BB$_1$(gen), …. *error* = =#BB$_N$(pre) and *error* == #BB$_N$(bran)} is introduced to describe the error occurrence. By defining each expression as a DEFINE statement, such as "h_i= (error == #BB$_j$(body)) ", the fault model, which indicates that only one error occurs per execution, can be expressed as the following INVAR statement:

$$h_1 + h_2 + h_3 + ... + h_w = 1$$

where *W* represents the number of Boolean expressions.

Step 6: Defining the property to be verified

The basic principle of model checking is that the model under validation should satisfy a desired property. The property, which characterizes signature-monitoring mechanisms, states that if a CFE disrupts the control flow execution, the error is ultimately detected. This property can be described in a CTL formula.

$$AG\ (h_1 + h_2 + h_3 + \ldots + h_W = 1 \rightarrow AF(PC = \texttt{fault\_detected}))$$

The CTL formula shows that in all state sequences from the initial state, if a CFE occurs, the program control flow is eventually transferred to the final state “fault detected”.

6. Case Studies

In this section, we apply the verification method to two reprehensive error detection algorithms: CFCSS and DSM.

6.1. Model Checking of CFCSS algorithm

CFCSS is a pure software method proposed by researchers in the Stanford University [11]. CFCSS has been widely used in safety-critical systems. In the CFCSS algorithm, each block is numbered and assigned a unique static signature. Two basic checking instructions, namely, “xor G, G, d” and “bne G, S, error” are inserted at the beginning of each basic block, where G denotes the dynamic signature and S denotes the static signature. For each block that has more than one predecessor, a checking instruction “xor G,G,D” is inserted after the instruction “xor G, G, d”, where D denotes the assistant signature. Finally, for each block that has more than one successor, a checking instruction “xor D,S_h,S_i” is inserted after the instruction “bne G, S, error”. Given the limited space, the details of the CFCSS algorithm are presented in [11].

According to the characteristic of checking instructions in CFCSS algorithm, basic blocks can be classified into the following kinds (seeing Fig. 4):

- one to one: a block has only one predecessor and each of its successors has only one predecessor.
- one to many: a block has only one predecessor and at least one of its successors has more than one predecessor.
- many to one: a block has more than one predecessor and each of its successors has only one predecessor.
- many to many: a block has more than one predecessor and at least one of its successors has more than one predecessor.

Let S_i be the static signature of basic block BB_i, and d_i be the signature difference which is calculated as $d_i = S_i \oplus S_j$ (BB_j is the successor of BB_i).

The added checking instructions in Fig. 4 are PowerPC instructions [15].

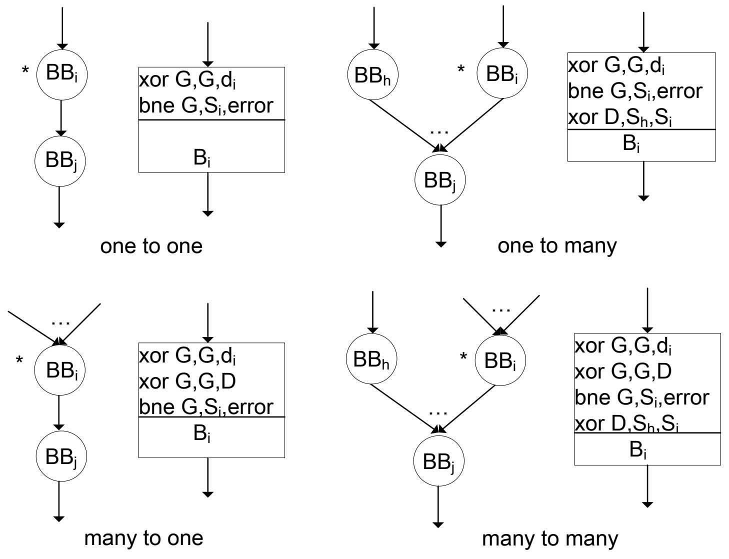


Fig. 4. The added checking instructions for different kinds of blocks

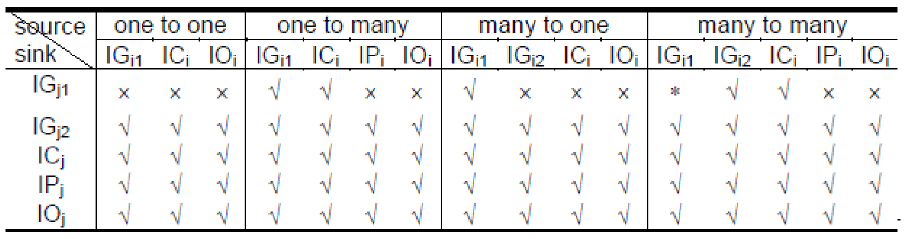
According to the verification, the results are shown in Table 1 and Table 2. They are obtained by statistically analyzing the counterexamples NuSMV reports. Before identifying the results, we introduce several notations. We use the notation *Suc*(BB_i) to denote the set of all successors of BB_i. Correspondingly, we use the notation *Pred*(BB_i) to denote the set of all predecessors of BB_i. We use the symbol "IG_{i1}" to denote the checking instruction "xor G, G, d_i" of BB_i. Similarly, the symbols "IG_{i2}", "IC_i", "IP_i" are used to denote the checking instructions "xor G, G, D", "bne G, S_i, error" and "xor D, S_h,S_j", respectively. And the symbol "IO_i" is introduced to represent the original instructions of BB_i.

A CFE can be modeling as a transition from the source instruction to the sink instruction. Table 1 lists the verification results for the CFEs between two blocks without any direct legal control flow transfers. In other words, BB_m is not in *Suc*(BB_i) and BB_i is not in *Pred*(BB_m). Table 2 lists the verification results for the CFEs between two blocks BB_i and BB_j, where $BB_j \in$ *Suc*(BB_i).

Table 1. Part I: the verification results of checking capacity for CFCSS .

source / sink	one to one			one to many				many to one				many to many				
	IG_{i1}	IC_i	IO_i	IG_{i1}	IC_i	IP_i	IO_i	IG_{i1}	IG_{i2}	IC_i	IO_i	IG_{i1}	IG_{i2}	IC_i	IP_i	IO_i
IG_{m1}	◊	√	√	◊	◊	√	√	√	◊	◊	√	√	◊	◊	√	√
IG_{m2}	√	√	√	√	√	#	#	∇	∇	∇	∇	∇	∇	∇	#	#
IC_m	√	√	√	√	√	√	√	∇	√	√	√	∇	√	√	√	√
IP_m	√	√	√	√	√	√	√	∇	√	√	√	∇	√	√	√	√
IO_m	◊	√	√	◊	◊	#	#	√	◊	◊	√	*	◊	◊	#	#

Table 2. Part II: the verification results of checking capacity for CFCSS .



source \ sink	one to one			one to many				many to one				many to many				
	IG_{i1}	IC_i	IO_i	IG_{i1}	IC_i	IP_i	IO_i	IG_{i1}	IG_{i2}	IC_i	IO_i	IG_{i1}	IG_{i2}	IC_i	IP_i	IO_i
IG_{j1}	×	×	×	√	√	×	×	√	×	×	×	*	√	√	×	×
IG_{j2}	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√
IC_j	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√
IP_j	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√
IO_j	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√

The symbol "√" indicates that the illegal transitions from the source instruction to the sink instruction can be checked by CFCSS algorithm, while symbol "×" indicates that the illegal transitions cannot be detected. Other symbols describe illegal transitions that cannot be detected under certain conditions. The undetected errors and proofs are obtained by analyzing the counter-examples reported by the model checker NuSMV. They are as follows:

– "×": When the dynamic signature G and the assistant signature D are updated at BB_i, control flow skips IO_i of BB_i and transfers to IG_{j1} of BB_j, where $BB_j \in$ *Suc*(BB_i). These illegal transitions cannot be detected.

– "#": Undetected errors are grouped into two cases: 1) when new G and new D are generated at BB_i, control flow transfers to IO_m of BB_m, where $BB_m \in$ *Pred*(BB_j) and $BB_j \in$ *Suc*(BB_i) and 2) when new G and new D are generated at BB_i, control flow transfers to IG_{m2} of BB_m, where $BB_m \in$ *Pred*(BB_j) and $BB_j \in$ *Suc*(BB_i).
  Proof. Suppose an illegal branch br_{im} takes from IO_i of BB_i to IO_m of BB_m, where $BB_m \in$ *Pred*(BB_j) and $BB_j \in$ *Suc*(BB_i). At BB_i, G is equal to S_i and D is equal to $S_i \oplus S_m$ (picked arbitrarily). After br_{im} is taken, the original instructions of BB_m are executed, and then control flow transfers to its successor BB_j. The checking instructions of BB_j update $G= G \oplus d_j \oplus D = S_i \oplus (S_j \oplus S_m) \oplus (S_i \oplus S_m) = S_j$. With G equal to S_j, the checking instruction "bne G, S_j, error" does not detect the mismatch. Therefore, the error cannot be detected. Other cases labeled by "#" can be proved in the same way.

– "*": An illegal transition taken from IG_{i1} of BB_i to IG_{j1} of its successor BB_j cannot be detected if and only if BB_i is a "many to many" block. Similarly, illegal transitions taken to IO_m of BB_m cannot be detected, where $BB_m \in$ *Pred*(BB_j).
  Proof. Suppose br_{ij} is an illegal branch, where $BB_j \in$ *Suc*(BB_i), $BB_h \in$ *Pred*(BB_i), and $BB_k \in$ *Pred*(BB_i). At BB_i, G is equal to $(S_h \oplus d_i)$ and D is equal to $(S_h \oplus S_k)$. After br_{ij} is taken, the checking instructions of BB_j update G, where $G = (S_h \oplus d_i) \oplus d_j \oplus D = S_h \oplus (S_i \oplus S_k) \oplus (S_i \oplus S_j) \oplus (S_h \oplus S_k) = S_j$. Therefore, this illegal branch is not detected. Illegal transitions taken from IG_{i1} to IO_m of BB_m can be proved similarly.

– "∇": Undetected errors of this kind are caused by the design of signatures. Suppose that $BB_h \in$ *Pred*(BB_i) and $BB_k \in$ *Pred*(BB_i), BB_m is a random block. If equation (10) is satisfied, illegal transitions taken from IG_{i1}

of BB_i to IC_m of BB_m or from IG_{i2} of BB_i to IG_{m2} of BB_m cannot be detected, where BB_m has more than one predecessor.

$$S_m = S_h \oplus S_k \oplus S_i \tag{10}$$

Proof. With br_{im} as an illegal branch, as aforementioned, and the signature design satisfies equation(10), after the instruction "xor G,G,d_i" of BB_i is executed, br_{im} is taken, where G is equal to $(S_h \oplus d_i) = S_h \oplus (S_k \oplus S_i)$. With S_m equal to $S_h \oplus S_k \oplus S_i$, the instruction "bne G, S_m, error" of BB_m cannot detect the error. The detection process of illegal transition from "xor G,G,D" of BB_i to "xor G,G,D" of BB_m is similar.

– Undetected errors of this kind are also caused by the design of assigned signatures. If equation (11) is satisfied, illegal branches taken from IC_i, where the new G at BB_i is generated, to IG_{m1} of BB_m are not detected, where BB_m has more than one predecessor. Similarly, illegal transitions to the original instructions of the predecessor BB_m also cannot be detected. D is determined by the block that is executed prior to BB_i.

$$S_m = S_i \oplus d_m \oplus D \tag{11}$$

Proof. With br_{im} as an illegal branch, as aforementioned, the signature design satisfies equation (11). After the new G of BB_i is generated, br_{im} is taken, where G is equal to S_i. The checking instructions of BB_m update G, where $G=S_i \oplus \oplus d_m \oplus D=S_m$. Therefore, br_{im} cannot be detected.

Oh et al. [11] evaluated the effectiveness of CFCSS by fault injection experiment and their experimental results show that the CFCSS algorithm increases the error detection capability by an order of magnitude. However, the results do not exactly specify which faults do not attempt detection. As of this writing, the present study is a pioneer evaluation of the effectiveness of the algorithm and the identification of undetected errors.

6.2. Model Checking of DSM algorithm

DSM is a software-based signature analysis technique presented by TMA laboratory [9]. In DSM, each basic block is associated to an identification number (IDB) and each interblock transition is assigned as:

$$\text{signature} = IDB_{source} \mid IDB_{destination} \tag{12}$$

where operator "|" represents the concatenation function.

In addition to the transition signatures, some local cumulative signatures N_{i1}, N_{i2}, N_{i3} are introduced. These signatures are integer numbers that are unique for each basic block. The condition expressed by (13) is satisfied by the design.

$$N = N_{11} + N_{12} + N_{13} = \ldots = N_{N1} + N_{N2} + N_{N3} = 0 \tag{13}$$

The local cumulative signatures ensure that: (1) a source block transfers control to the first instruction of the destination block and (2) the signature-checking instructions are correctly executed. Fig. 5 demonstrates how control flow is checked [9], where *B* denotes the dynamic transition signatures and *R* denotes the static transition signatures. The details of DSM algorithm refer to [9].

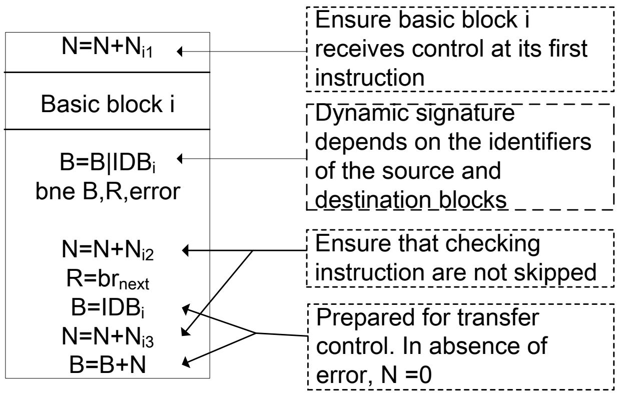

$N=N+N_{i1}$
Basic block i
$B=B|IDB_i$
bne B,R,error
$N=N+N_{i2}$
$R=br_{next}$
$B=IDB_i$
$N=N+N_{i3}$
B=B+N
Ensure basic block i receives control at its first instruction
Dynamic signature depends on the identifiers of the source and destination blocks
Ensure that checking instruction are not skipped
Prepared for transfer control. In absence of error, N =0

Fig. 5. The implementation of DSM algorithm

By applying the proposed method to validate DSM algorithm, the verification results are listed in Table 3. Notations "IL_{i1}", "IL_{i2}" and "IL_{i3}" are introduced to represent the instructions "$N=N+N_{i1}$", "$N=N+N_{i2}$" and "$N=N+N_{i3}$", respectively. "IO_i" is used to denote the original instructions of BB_i. "IG_i" denotes the signature generation instruction "$B=B|IDB_i$". "IC_i" denotes the compare instruction "bne B,R,error". "IP_{i1}", "IP_{i2}" and "IP_{i3}" are used to denotes the prepare instruction "$R=br_{next}$", "$B=IDB_i$" and "B=B+N", respectively. In Table 3, there are no legal control flow transitions between BB_m and $BB_{i.}$

Table 3. the verification results of checking capacity for DSM.

source / sink	IL_{i1}	IO_i	IG_i	IC_i	IL_{i2}	IP_{i1}	IP_{i2}	IL_{i3}	IP_{i3}
IL_{m1}	√	√	√	√	√	√	#	#	√
IO_m	√	√	√	√	√	√	√	√	√
IG_m	√	√	√	√	√	√	√	√	√
IC_m	√	√	*	*	*	√	√	√	√
IL_{m2}	√	√	√	√	◊	◊	◊	√	√
IP_{m1}	∇	∇	∇	∇	√	√	√	√	√
IP_{m2}	√	√	√	√	√	√	√	√	√
IL_{m3}	√	√	√	√	√	√	√	√	√
IP_{m3}	√	√	√	√	√	√	√	√	√

In [4] and [9], DSM algorithm is validated to have the ability to detect all the illegal interblock transitions. However, we found four kinds of errors that cannot be detected under certain conditions. These undetected errors are as follows [14]:

- "#": Illegal branches that are transferred from one of the instructions between "B=IDB$_i$" and "N=N+N$_{i3}$" of BB$_i$, to the instruction "N= N+N$_{n1}$"of BB$_n$ cannot be detected, where BB$_i$ ϵ *Pred*(BB$_n$) and *Suc*(BB$_n$) ϵ ∅ . These faults do not cause wrong outputs and can thus be negligible.
- "∗": Illegal branches taken from one of the instructions "B=B|IDB$_i$; bne B, R, error" of BB$_i$ to the instruction "bne B, R, error" of BB$_n$ cannot be detected, where *Suc*(BB$_n$) ϵ ∅.
  Proof. Assuming that br$_{in}$ is an illegal branch and the branch is transferred from "B=B|IDB$_i$" of BB$_i$ and landed at "bne B, R, error" of BB$_n$, where *Suc*(BB$_n$) ϵ ∅, when br$_{in}$ is taken, R is still equal to B because the new R is not generated at BB$_i$. Therefore, they match and br$_{in}$ is not detected.
- "∇": Undetected errors of this kind are caused by the design of local cumulative signatures. If equation (14) is satisfied, illegal branches are transferred from one of the original instructions of BB$_i$ to the checking instruction "R=br$_{next}$" of BB$_m$, which are cannot be detected. BB$_i$ and BB$_m$ are arbitrary basic blocks of the program.

$$N_{i1}+ N_{m3} = 0 \qquad (\mathbf{14})$$

  Proof. If br$_{im}$ is an illegal branch and it is taken from one of the original instructions of BB$_i$, skipped the rest of the checking instructions, and then landed at the instruction "R=br$_{next}$" of BB$_m$. At BB$_i$, N is equal to N$_{i1}$. After br$_{im}$ is taken, the new R is generated to the transition signature, depending on the identifiers of BB$_m$ and its successor. B is updated to the identifier of BB$_m$ and N is added by N$_{m3}$. If equation (14) is satisfied, N equals to zero. The control flow then transfers to the successor of BB$_m$. Considering that R and B are updated at BB$_m$, the execution of the program continues without detecting br$_{im}$.
- "◊": Undetected errors of this kind are also caused by the design of local cumulative signatures. If equation (15) is satisfied, illegal branches that are transferred from the instructions "N=N+N$_{i2}$" of BB$_i$ to the checking instruction "N=N+N$_{m2}$" of BB$_m$ are not detected. BB$_i$ and BB$_m$ are arbitrary blocks of program.

$$N_{i1}+ N_{i2} + N_{m2}+ N_{m3} = 0 \qquad (\mathbf{15})$$

  Proof. If br$_{im}$ is an illegal branch, as aforementioned, and the signature design satisfies equation (15). After the instruction "N=N+N$_{i2}$"of BB$_i$ is executed, br$_{im}$ is taken, where N is equal to (N$_{i1}$ +N$_{i2}$). At BB$_m$, the checking instructions generate R and B. N is updated as N= N+N$_{m2}$ +N$_{m3}$ = (N$_{i1}$ +N$_{i2}$) +N$_{m2}$ +N$_{m3}$ =0. Given that equation (15) is satisfied, no error is detected when the control flow is transferred to the successor of BB$_m$.

7. Comparisons

As illustrated in Section 2, some earlier works have proposed the use of model checking in evaluating system dependability. The work with more similarity to ours is described in [4], but ours is much more general and practical. The comparisons are as follows:

- We provide a general approach to evaluate all signature monitoring techniques, while the technique in [4] is just aimed at the DSM technique.
- In [4], the model checking computation time increases at a polynomial rate of the addition of instructions of the model, and the amount of memory needed is a potential bottleneck for large models. It is known as the state-space explosion problem. In this study, we offer a solution to solve this problem. The states variables consist of (*PC*, *S*, *Con*, *error*), so the state space increases at a polynomial rate of the addition of the basic blocks for a given program. Therefore, the memory cost and computation time can be seen as a biquadrate function of *N*, where *N* denotes the number of basic blocks. Apparently, the number of basic blocks is much less than the number of instructions for a given program. So our approach is more efficient in optimizing the state space exploration. From the viewpoint of model checking, our approach avoids the state-space explosion problem.
- In contrast with the verification results in [4], our verification results can be used to design signatures as the signature variables are involved in the model. For instance, to dissatisfy the condition expressed by "N_{i1} + N_{i2} + N_{m1} +N_{m2} =0", illegal transitions symbolized by "◊" in Table 3 can be detected by DSM algorithm.

8. Conclusions

In this paper, we presented a novel approach for evaluating the effectiveness of signature-monitoring techniques. We initially modeled the program that was strengthened by signature-monitoring algorithms as a control flow machine. The execution of the assembly program modeled by the control flow machine is specified using a step-operational semantics, which maps a machine state to other machine states. We then refined the control flow machine into a state transition system and proposed a translation procedure to translate the state transition system into the input program of the model checker NuSMV for automatic verification. Finally, we applied the approach to two reprehensive error detection algorithms, namely, CFCSS and DSM. The undetected errors were analyzed based on the counter-example reported by NuSMV. As of this writing, this is a pioneer evaluation of signature-monitoring techniques, where undetected errors are also uncovered for the first time.

In our future research, we shall improve the algorithm design according to the verification results. We shall also design a lexical analyzer which can translate the state transition system into SMV programs automatically.

References

1. Gnesi S., Lenzini G., Latella D., Abbaneo C., Amendola A., and Marmo P.: An Automatic SPIN Validation of a Safety Critical Railway Control System. In Proceedings of the International Conference on Dependable Systems and Networks, New York, NY, USA, 119-124. (2000)
2. Ramachandran P., Kudva P., Kellington N.: Statistical Fault Injection. In Proceedings of the 38th International Conference on Dependable Systems and Networks. Conference Publishing Services, Anchorage, USA, 122-127. (2008)
3. Kljaich J., Smith B. T., Wojcik A. S. : Formal Verification of Fault Tolerance Using Theorem-Proving Techniques. IEEE Transaction on Computers, Vol. 38, No. 3, 366-376. (1989)
4. Nicolescu B., Gorse N., Savaria Y.: On the Use of Model Checking for the Verification of a Dynamic Signature Monitoring Approach. IEEE Transactions on Nuclear Science, Vol. 52, No. 5, 1555-1561.(2005)
5. Tomoyuki Y., Tatsuhiro T., Tohru K.: Automatic verification of Fault Tolerance Using Model Checking. In Proceedings of the 8th Pacific Rim International Symposium on Dependable computing. Conference Publishing Services, Seoul, Korea 95-102. (2001)
6. Roberto C., Alessandro C., Charles A. J. , Gavin K., Emanuele O., Pistore M. , Roveri M., Andrei T.: NuSMV 2.4 User Manual. [Online]. Available: http://nusmv.fbk.eu/NuSMV/userman/v24/nusmv.pdf (current 2005)
7. Vemu, R., Gurumurthy, S., Abraham J. A.: ACCE: Automatic correction of control-flow errors. In Proceedings of 2007 International Conference on TEST. Conference Publishing Services, Santa Clara, USA, 1-10. (2007)
8. Goloubeva O., Reaudengo M., Sonza Reorda M., Violante M.: Soft-Error Detection Using Control Flow Assertions. In Proceedings of the 18th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems. Boston, USA, 581-588. (2003)
9. Nicolescu B., Savaria Y., Velazco R. : Software detection mechanisms providing full coverage against single bit-flip faults. IEEE Transaction on Nuclear Science, Vol. 51, No. 6, 3510-3518. (2004)
10. Borin E., Wang C., Wu Y.F., Araujo G. : Software-Based Transparent and Comprehensive Control-Flow Error Detection. In Proceedings of the 4th ACM/IEEE International Symposium on Code Generation and Optimization. New York, USA. (2006)
11. Oh N. , Shirvani P.P., McCluskey E.J.: Control-Flow Checking by Software Signatures. IEEE Transactions on Reliability, Vol. 51, No. 2, 111-122. (2002)
12. Aho A.,Sethi R., Ullman J.: Compilers: Principles, Techniques and Tools(2nd Edition). Addison-Wesley, Boston, USA.(2007)
13. Reinhardt S. K., Mukherjee S. S.: Transient fault detection via simultaneous multithreading. In Proceedings of the 27th Annual International Symposium on Computer Architecture. ACM Press, Vancouver, Canada.(2000)
14. Tan L., Tan Q., Xu J., Li J.: A note on “On the Use of Model Checking for the Verification of a Dynamic Signature Monitoring Approach. IEEE Transaction on Nuclear Science, Vol. 58, No. 1, 359-359. (2011)
15. Programming Environments Manual for 32-Bit Implementations of the PowerPC™ Architecture. [Online]. Available: www.freescale.com (current 2005)
16. Arora A. and Gouda M.: Closure and Convergence: A Foundation of Fault-Tolerant Computing. IEEE Transactions on Software Engineering, Vol.19, No.11, 1015–1027. (1993)

17. John R.: Systematic Formal Verification for Fault-Tolerant Time-Triggered Algorithms. IEEE Transactions on Software Engineering, Vol.25, No.5, 651–660. (1999)
18. Daniel L. and Ruben A.: Formal Verification of Fault Tolerance Aspects. [Online]. Available: http://www.software3.net/f/formal-verification-of-fault-tolerance-aspects-w290 (current 2005)
19. Sexton F. W.: Destructive single-event effects in semiconductor devices and ICs. IEEE Transactions on Nuclear Science, Vol. 50, No. 3, 603–621. (2003)
20. Meng Z., Anita L. and Daniel J. S.: Analyzing Formal Verification and Testing Efforts of Different Fault Tolerance Mechanisms. In Proceedings of the 24th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, Chicago, IL, USA, 277-285.(2009)

Lanfang Tan received the B. S. degree from the Department of Computer Science, National University of Defense Technology in 2008. She is currently pursuing the Ph.D. degree. Her research interests include software fault-tolerance, formal verification, model checking and dependable computing.

Qingping Tan received his M.S. and Ph.D. degrees in computer science from the National University of Defense Technology in 1988 and 1992, respectively. He is currently a professor and Ph. D. supervisor at National University of Defense Technology. He is also a visiting scholar at the University of Pisa in 2009. His research interests include software engineering, programming languages, compilers, software fault-tolerance and dependable computing.

Jianjun Xu received his M.S. and Ph.D. degrees in computer science from the National University of Defense Technology in 2006 and 2010, respectively. He is a full-time Lecturer in the Computer Science School at the National University of Defense Technology. His research is focused on program analysis, compiler for fault tolerance.

Huiping Zhou is an Associate Professor in the Computer Science School at the National University of Defense Technology. His research is focused on programming languages, compilers and software fault-tolerance.

Received: February 18, 2012; Accepted: November 26, 2012.