Signature-based Method for Run-Time Fault Detection in Communication Protocols

G. Noubir, K. Vijayananda, and H. J. Nussbaumer, IEEE Fellow
Swiss Federal Institute of Technology, Lausanne,
Computer Engineering Department, EPFL-DI-LIT,
CH-1015, Lausanne, Switzerland
{noubir,vijay,nussbaumer}@di.epfl.ch

Abstract

Run-time fault detection of communication protocols is essential because of faults that occur in the form of coding defects, memory problems and external disturbances. We propose a signature-based method to detect run-time faults. A *polynomial* using the state and event information as coefficients is used to transform a sequence of states and events into a number (signature). The static signature corresponding to the correct execution of the protocol is compared with the run-time signature. This technique is reliable, fast, and efficient compared to the existing techniques. The states and events are assigned values such that multiple paths leading to the same state result in a unique signature. This reduces the number of run-time comparisons required to verify the correct execution of the protocol. Fault-detection based on signatures is also much simpler than observer-based methods. We propose extensions to communication protocols that facilitate the application of signature-based techniques to detect run-time faults in communication protocols. In this paper, we present *eXTP4*, an extended transport layer protocol that facilitates run-time fault detection.

Keywords

Fault detection, communication protocol, protocol faults, signature, polynomial, observer.

INTRODUCTION

Communication protocols are designed to provide reliable services for meaningful exchange of information. Protocol verification and validation techniques are used to ensure the correctness of their specification. However, there is no strategy to ensure that the execution of the protocol is error-free. Dormant errors due to coding, memory problems, and external disturbances are not considered in above mentioned techniques. Unless some kind of run-time checking is done, this type of faults may remain undetected until a partial or complete breakdown of the communication system. In order to ensure the continuous availability and quality of services provided by these systems, it is necessary to detect and diagnose run-time faults in communication protocols.

Several techniques for detecting run-time faults in programs can be found in the literature. These techniques include *observer-based* methods for communication protocols and *signature-based* methods for programs. The observer-based methods uses an external observer and a model of the communication protocol to detect faults [1, 2, 3, 4]. At run time, an external observer is used to monitor the messages exchanged between entities. These messages are compared with the model for their correctness. Signature-based methods are a very popular technique used to

verify the run-time behavior of programs [5, 6, 7, 8]. They are used at different levels to monitor and detect run-time faults and to check the control flow of programs. Each legal path in the control flow graph of the program is transformed into a signature which is referred to as the static signature. The static signature corresponds to the correct execution of the program. The signature is again computed during the execution of the program and this is compared with the static signature to verify that the program has traversed a legal path.

In this paper, we combine these two techniques for the purpose of detecting run-time faults in communication protocols. An external observer uses signatures to verify the execution of communication protocols. A novel function based on polynomials is used to generate the signature. Our signature-based approach is a probabilistic method which can be used to detect run-time faults in communication protocols. We would like to emphasize that after the fault has been detected using this technique, other techniques have to be used for a detailed diagnosis in order to localize the fault and initiate recovery procedures. The signature-based method may aid the diagnosis and recovery by pinpointing the faulty event and the state of the communication protocol when the fault occurred. We show that this method reduces the complexity of the observer compared to other observer-based methods. The low complexity of the proposed method makes it tractable for run-time fault detection in communication protocols.

The rest of the paper is organized as follows. Research done in the related domain is first discussed. We then present the definitions and terminologies used in this paper. In the next section, we discuss the fault detection technique. Then, we present our signature generation technique. In the following section, we describe the use of a new signature-based method for run-time fault detection in communication protocols. An example from Transport Class 4 (TP4) is used to illustrate the fault-detection mechanism. Then, the extensions to the communication protocols that facilitate this new fault-detection scheme are introduced. Finally, Transport Class 4 protocol is used to describe some implementation details of the proposed extensions.

STATE OF THE ART

We now briefly present the related work in the area of observer-based and signature based methods.

Observer-based Methods: Several variations of the observer-based method can be found in the literature. In [1], Bouloutas et al. use the Finite State Machine (FSM) approach to detect faults in machines whose behavior is described using FSMs. Observers, which are modeled using FSMs, are used to detect out-of-sequence messages and incorrect state transitions. In [2], Wang et al. decompose the FSM representing the communication protocol into several FSMs. Each FSM is represented by an observer. All the observers operate in parallel to detect the faults in the communication protocols. In [4], Diaz et al. use an observer to monitor and detect faults in an implemented system. The observer is modeled using the specifications of the implemented system. In [9, 10], Oikonomou uses an observer based on the abstraction of the FSM model to detect faults in the system. The abstracted observer uses the abstracted FSM model of the system to detect deviations in the behavior of the system. In [11, 3], Riese uses model-based reasoning to detect deviations from the correct behavior of the communication protocols. A FSM model of the communication protocol is used by the observer to diagnose the faults.

In most of the observer-based methods, the model of the communication protocol used by the observer is as complex as the communication protocol itself. In the work proposed by Wang et al. and Oikonomou the complexity of the observer is reduced compared to the communication protocols. However these methods suffer from other problems like non-determinism and error latency [1]. In this paper, we propose to use a signature-based method to detect run-time faults. We combine the observer-based method and the signature-based method to detect faults. We show that this method reduces the complexity of the observer compared to other observer-based methods.

Signature-based Methods: In [5], Yau et al. use the path-transformation method to generate signatures. Prime numbers are assigned to every event in a loop-free path and the signature is computed as the product of all the prime numbers in the path. The signature of a path traversed during run-time is compared against a table containing all the legal path signatures. This method cannot detect out-of-sequence events because of the commutative and associative property of multiplication of prime numbers. In [6], Saxena et al. use an extended precision checksum as a signature to verify run-time execution of the program. A checksum is assigned to every block of sequential code. The generation and verification is done in hardware. A watchdog monitor is used to verify the correct execution of every block of code. This method cannot detect out-of-sequence events and requires hardware support for verification. In [7], Upadhyaya et al. use a signature which is a *m-out-of-n* code. A known signature technique is applied to an instruction stream at compile time and when the accumulated signature forms a m-out-of-n code, the instruction is tagged. At run-time, the signature is accumulated and when a tagged instruction is reached, a check is made to see if the current signature forms a m-out-of-n code. In this method, there is no control over the error latency. It requires hardware support and is well-suited for self-monitoring processes and is not suitable for detecting faults in communication protocols. In [8], Leveugle proposes a method for hardware control flow checking of sequential circuits defined by their Mealy machine. The signature is computed using Multiple Inputs Shift Registers (MISR). Leveugle presents an algorithm to solve the state assignment problem when possible. This method requires a restricted set of graphs (SC graph) and hardware support to compute and verify the signature. Also, no algebraic method is available to solve the state assignment problem. It is not well-suited for monitoring communication protocols. Leveugle uses the concept that paths leading to the same state must have the same signatures and in this paper we extend this idea to a more general signature function to reduce the total number of signatures and an algebraic solution to the event-state assignment problem.

To summarize, most of signature-based methods reviewed here have been applied in the domain of hardware and hence they require support to generate and verify the signature. The observer is part of the program under observation and is within the environment of the program. The fault coverage is restricted and some of them suffer from a larger error detection latency. Even if these methods are implemented in software, it will be necessary to modify the program. These methods are well suited to monitor programs. An external observer cannot use these methods to detect run-time faults. Hence, they need to be modified if they are to be used for detecting run-time faults in communication protocols.

[1] Error latency is defined as the time between the occurrence of a fault and its detection.

Our Method and Contributions

We propose a new signature-based method to detect run-time faults in communication protocols [12, 13, 14]. It does not require hardware support and is simple, fast and efficient compared to other methods. This method uses a *polynomial* to transform a sequence of states and events into a number (signature). The signature is computed using the *state* and the *event* information. The *static signature* which corresponds to the correct sequence of messages, is computed and stored in a table. During the execution of the communication protocol, an *external observer* computes the *run-time signature* and compares it with the static signature. A fault is detected when the run-time signature differs from the static signature.

Run-time signature is computed in an efficient manner using Horner's rules (step-wise computation). This method generates distinct signatures that can detect *illegal transitions* and *out-of-order* events. Hence, this method is more reliable than other existing methods, that cannot detect out-of-order events. The state and events are assigned values such that correct paths leading to the same state have the same signature. This reduces the number of comparisons required to verify the correctness of the run-time path. In this paper, we show that the complexity of the observer using the signature-based method is reduced compared to the observer-based methods or Model-based methods. Moreover, the environment of the external observer is different from the program. Hence, it is not affected by the perturbations of the program's environment.

In the past, communication protocols have been modified for the purpose of improving their performance [15, 16, 17]. In this paper, we propose extensions (without modifications to the protocol) to communication protocols to facilitate the detection of run-time faults using our signature-based method. Extensions are proposed to include the state information in the messages exchanged between communication entities. Different methods for the computation of the run-time signature and its comparison with the static signature are analyzed based on the extensions to the communication protocols.

DEFINITIONS AND TERMINOLOGY

In this section, we recall some definitions of finite state machines, paths, correct paths, legal state paths, and legal event paths.

Definition 1 *The communication protocol is modeled as an FSM. An FSM is a 3-tuple* $A = (Q, \Sigma, \delta)$*, where* Q *denotes the set of all possible states* $\{S_1, \ldots, S_n\}$*.* Σ *is the set of all possible events. In this paper, the term event and message are used interchangeably to denote the messages exchanged between entities.* δ *denotes the state transition function* ($\delta : Q \times \Sigma \rightarrow Q$)*.*

Each state S_i (event E_i) is associated with a value s_i (e_i) from the working algebraic field F. In general, F is a *Galois* field (i.e., finite field).

Definition 2 *A Galois field* $GF(p)$ *is a field with a finite number* (p) *of elements.* p *is a power of a prime number. When* p *is a prime number addition and multiplication in* $GF(p)$ *are done modulo* p*. In this paper we consider only Galois fields with a prime number of elements. In [18],*

we proved that signatures computed using a MISR are a special case of the signature function we propose, where the Galois field is an extension field $GF(2^k)$.

Definition 3 *A path C is defined as an alternating sequence of states S_i and events E_j (e.g., $S_1E_1S_5E_4 \cdots E_{n-1}S_n$).*

The state path *is the sub-sequence of states derived from the path by deleting all the events and retaining the states in their original order (e.g., $S_1S_5S_3 \ldots S_n$).*

The event path *is the sub-sequence derived from the path by deleting all the states and retaining all the events in their original order (e.g., $E_1E_4 \cdots E_{n-1}$).*

Definition 4 `Last(`C`)` *of a path C is the last state in C, (e.g., $last(S_2S_1S_7S_3) = S_3$).* `First(`C`)` *of a path C is the first state in C (e.g., $first(S_2S_1S_7S_3) = S_2$).* `Penultimate(`$C$`)` *is the last but one state in a path (e.g., $Penultimate(S_2S_1S_7S_3) = S_7$).*

Definition 5 *A path C is* correct *if and only if for every sub-sequence $S_iE_jS_k$ of C; $\delta(S_i, E_j) = S_k$.*

A state path $C : S_1S_5S_3 \cdots S_n$ is a legal state path if and only if for every sub-sequence S_iS_j of C, there exists an event $E \in \Sigma$ such that $\delta(S_i, E) = S_j$. Otherwise, the state path is an illegal state path.

An event path $C : E_1E_3E_2 \cdots E_n$ is a legal event path *if and only if for every sub-sequence E_iE_j of C, there exists three states $(S_k, S_l, S_m) \in Q^3$ such that $\delta(S_k, E_i) = S_l$ and $\delta(S_l, E_j) = S_m$. Otherwise, the event path is an illegal event path.*

METHODOLOGY

In this section, we provide the rational behind the signature-based method and describe the run-time strategy to detect faults in communication protocols. A fault is detected as a deviation from the correct behavior.

The concept of signatures

Consider the FSM representation of a program shown in figure 1(a). $S_1 \ldots S_{10}$ represent the states of the program. (a, b, c, d, e) represent the set of events that cause the program to change from one state to another. The execution of the program is represented by a sequence of states and events. This also known as the `path`. The simplest way to verify the correct execution of the program is to store all the correct paths in a table. If we assign values to the states and events, then the path can be stored as a sequence of numbers. Now one can compare the run-time path with all the correct paths in the table to verify its correctness.

For example, consider the FSM shown in figure 1(b). Let the states be assigned the values $1, \ldots, 10$, i.e. $(S_1, \ldots, S_{10}) = (1, \ldots, 10)$ and the events be assigned the values $(a, b, c, d, e) =$ (11, 12, 13, 14, 15). Then the path $C_1 = S_1bS_8dS_3eS_7$ can be represented by the following sequence of numbers: $(1, 12, 8, 14, 3, 15, 7)$

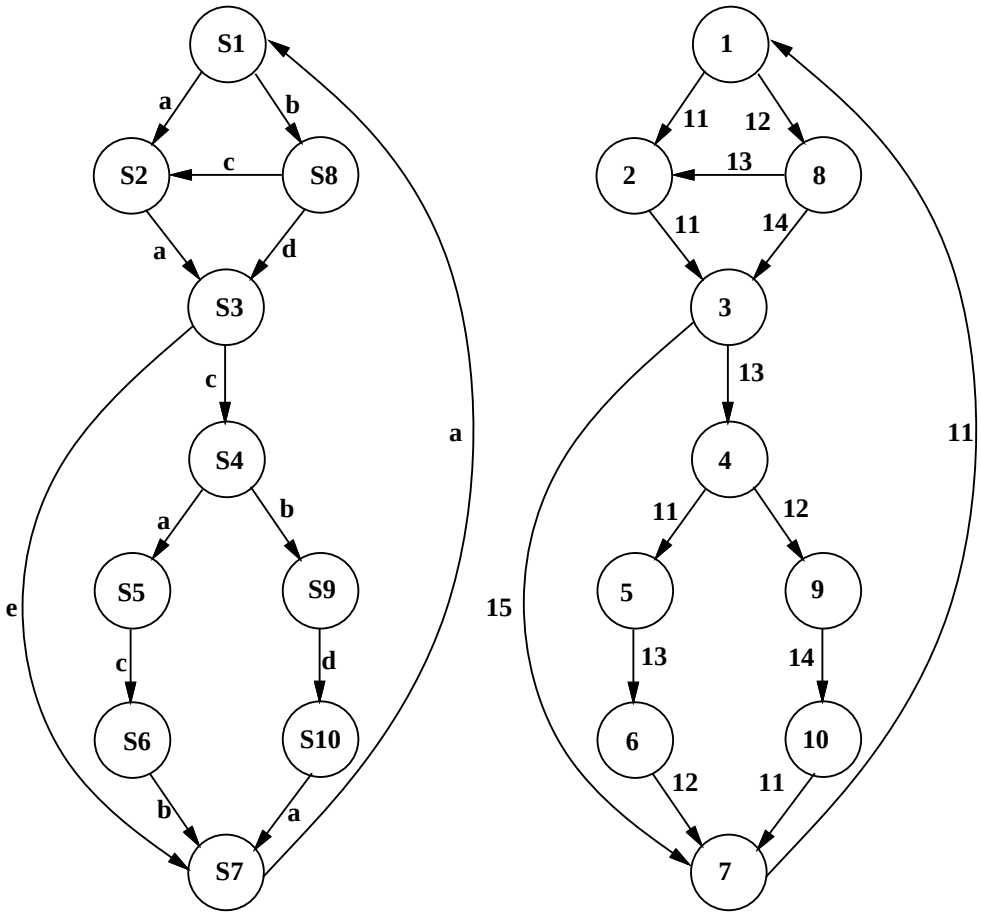

Figure 1: (a) FSM representation of a program (b) FSM with state and event values

Now this leads to two problems. First, the number of correct paths can potentially be very large. This requires large memory to store all the correct sequence of numbers in a table. In the above figure, there are potentially infinite paths which represent the correct behavior of the program. Second, this sequence of numbers do not have any mathematical structure and properties. Hence it is difficult to reason about them, analyze and study their properties.

In order to overcome these problems, we must choose a better representation so that the sequence of numbers have a better mathematical structure. This representation must also reduce the size of the table that contains all the correct paths. We could use hashing techniques to store and access these sequence of numbers. Similar techniques are used by compilers to store symbols, variables and access them at run-time. This solves the problem of storing and accessing large sequence of number. However, they still do not have any well defined mathematical structure. We look at some of the techniques used in digital signal processing to represent digital signals [19, 20]. One way is to add weight to every state and event in the sequence. This can be achieved by using the state and event values as coefficients of a polynomial. For example the path $C_1 = S_1 b S_8 d S_3 e S_7$ can be represented as

$$S_1 * x^6 + b * x^5 + S_8 * x^4 + d * x^3 + S_3 * x^2 + e * x + S_7 \quad (1)$$

Substituting the values of the states and events in 1, we get the following polynomial that represents the path $C_1 = S_1 b S_8 d S_3 e S_7$:

$$1 * x^6 + 12 * x^5 + 8 * x^4 + 14 * x^3 + 3 * x^2 + 15 * x + 7 \quad (2)$$

When the program changes from state S_7 to S_1 after the occurrence of the event a, the new path is $C_2 = S_1 b S_8 d S_3 e S_7 a S_1$. The polynomial representation of the new path is given as follows:

$$S_1 * x^8 + b * x^7 + S_8 * x^6 + d * x^5 + S_3 * x^4 + e * x^3 + S_7 * x^2 + a * x + S_1 \quad (3)$$

$$1 * x^8 + 12 * x^7 + 8 * x^6 + 14 * x^5 + 3 * x^4 + 15 * x^3 + 7 * x^2 + 11 * x + 1 \quad (4)$$

Now the polynomial representation has a better structure. If we evaluate the polynomial at a given point x_0, we can transform the path into a number. Taking $x_0 = 2$, the path C_1 represented by the polynomial in Eq. (2) evaluates to 737 and the path C_2 represented by the polynomial in (4) evaluates to 2971. Thus an entire sequence of numbers can be represented by a single number. The polynomial representation is much better than a sequence of numbers and this requires less space for its representation.

The next question to be answered is how can such a polynomial be used to verify the correct execution of a program. Every path has an associated polynomial and can be transformed into a number by evaluating the polynomial at a given point x_0. This number is known as the `signature` of the path. We store the signatures of all the correct paths in a table. At run-time, we compute the signature for the run-time path and compare it with the correct path signatures stored in the table.

This still does not solve the problem of potentially infinite number of correct paths corresponding to a program. This method still requires a large amount of memory to store the signatures of all the correct paths. We can solve this problem by associating the signature of a path with the last state in the path. When more than one path leads to the same state, the values of the states and events in the path are chosen such that all the paths leading to the same state have the same signature. Thus every state has a unique signature. In the later part of the paper, we show how this condition is used to assign values to the states and events. This is known as the `event-state assignment problem`. For example, the last state of path C_2 is S_1. Hence the signature of state S_1 is 2809. We refer to the signature of a state as `static signature`. The table containing the static signature of all the states is known as the `static signature table`. The number of signatures are reduced to the number of states of the FSM.

If we carefully examine equation 3, we can see that it can be rewritten in the following manner:

$S_1 + a * x + (\mathbf{S_7} + \mathbf{e} * \mathbf{x} + \mathbf{S_3} * \mathbf{x^2} + \mathbf{d} * \mathbf{x^3} + \mathbf{S_2} * \mathbf{x^4} + \mathbf{b} * \mathbf{x^5} + \mathbf{S_1} * \mathbf{x^6}) * x^2$

This is equivalent to:

$$Signature(C_2) = Signature(C_1) * x^2 + a * x + S_1 \quad (5)$$

Equation 5 demonstrates two important things

- The signature of a state contains information about all the paths leading to a state.

- The signature of a path C_i can be recursively computed in terms of the signature of the subpath that is contained in C_i. In equation 5, the path C_1 is contained in path C_2. Thus the computation of the signature of a path requires only two multiplication and two additions for each state transition.

So far we have not discussed how to assign the values to the state and events and the choice of the point x_0 for evaluating the polynomial that represents a path. Ideally, we require that every path must be transformed into a unique number. Later on, in the paper, we discuss in detail about these requirements and we present the solution to the event-state assignment problem, and the choice of x_0.

Signature-based method

At run time, messages are exchanged between communication entities. Let $E_1, E_2, \ldots, E_k$ be a sequence of messages exchanged, where $E_1, E_2, \ldots, E_k \in \Sigma$. Let S_0 be the initial state of the communication protocol, and $S_1, S_2, \ldots, S_k$ be the subsequent states of the communication protocol such that $\delta(S_{i-1}, E_i) = S_i, i = 1, \ldots, k$. The states and messages are assigned values from a Galois field F such that it satisfies some of the constraints to reduce the number of comparisons. We denote this problem as the *state and event-assignment*. The path represented by $S_0E_1S_1E_2S_2, \ldots, E_kS_k$ is transformed into a number using a signature function Φ. This function is a polynomial with the state and event values as coefficients.

$\Phi : \Pi \rightarrow F$

where F represents the Galois field that represents the signature space and Π represents the set of all possible paths.

The signature is computed by evaluating the polynomial at a given point x_0 in the Galois field F. The correct behavior of the communication protocol is represented by the set of all correct paths: Π_C that start at state S_0. The signature corresponding to a correct path is known as the *static signature*. This signature is associated with the last state in the correct path. Furthermore, all correct paths leading to the same state have the same signature.

Fault Detection

Given the Π_C of a communication protocol, it is possible to detect run-time faults in a communication protocol using the signatures. The run-time path which includes the sequence of messages exchanged between the entities and the intermediate states, is transformed into a signature using Φ. This represents the run-time signature. Let S_l be the last state in the run-time path. The run-time signature is compared with the static signature of S_l. A fault is detected when the run-time signature does not match the static signature of S_l. This comparison is not restricted to the last state in the path. It can be done for every state in the path. Later in this paper, we discuss more about optimizing the frequency of the comparison.

Observer

An *observer* is used to detect run-time faults. The observer computes the run-time signature and compares it with the static signature for correctness. When the observer is internal (i.e, it is a part of the program [2]), we have a self-checking program. However, the observer and the program are subject to the same perturbations. In the case of an external observer, the environment [3] of the observer is different from that of the program. Hence, it is not affected by the perturbations of the program's environment. In this paper, we address the issues related to the external observer. It is worth mentioning that the FSM model of the communication protocol used by an external observer is based on the events of the protocol that are visible to the observer. This model may be different from the one used by the implementation of the communication protocol.

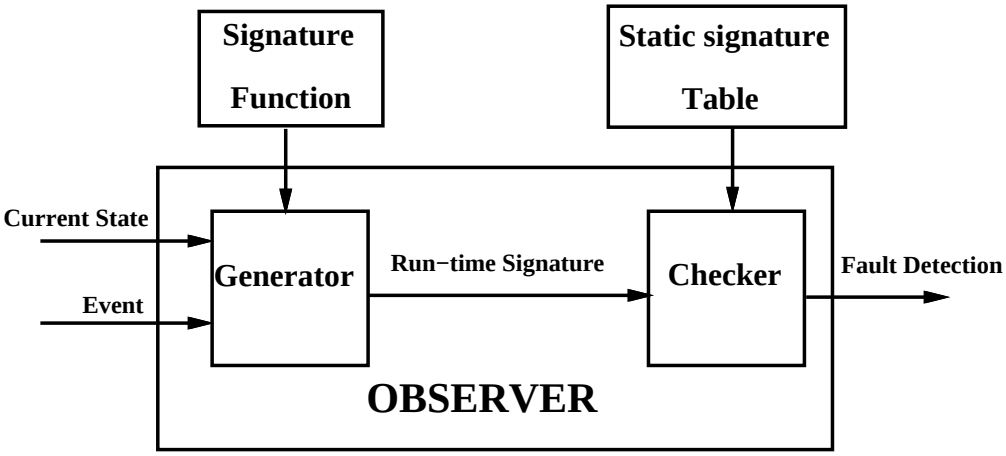

Figure 3: Observer Module

SIGNATURE GENERATION

In this section, we describe the signature generation technique that is used to transform a sequence of messages into a signature. A polynomial with the state and message information as coefficients is used to compute the signature. We also describe an algorithm that assigns values to states and events.

Requirements

Given a communication protocol modeled by a FSM, we would like to detect the illegal paths. The detection is done by transforming the run-time path[4] into a single value called *path signature* and comparing it with a static value computed off-line. The signature function Φ which is used in the transformation must have the following property:

$$\exists p < 1; Probability[\Phi(C_1) = \Phi(C_2) | C_1 \neq C_2] < p \tag{6}$$

[2] In this paper, the word *program* refers to the implementation of the communication protocol.
[3] The environment of a program includes the hardware and the software required to execute the program.
[4] The path can be a state, event, or full path.

p is called the *masking probability* and a good signature function has a low masking probability.

Formally, the problem is stated as follows:

Given a FSM $A = (Q, \Sigma, \delta)$ and a state S_0 with predefined signature equal to zero[5] such that all the other states are reachable from S_0, we require that the signature function must map every path beginning at state S_0 onto a unique value from an algebraic field F. In the following section, we introduce a signature-generation technique that satisfies property (6).

Path signature

We present three kinds of signatures: the *full-path*, the *state*, and the *event* signature. The main idea is to associate a path with a univariate polynomial defined over a given algebraic field F. The signature of a path C is the evaluation of its associated polynomial at a given point x_0 .

Choice of the signature function

A polynomial using the state and event information is used as the signature function. We choose a polynomial signature function for the following reasons:

1. Polynomials are well studied mathematical functions and their properties are well understood.
2. It is easy to compute the masking probability p defined in equation 6.
3. The state and event-assignment problem can be solved algebraically.
4. The signature can be computed in a step-wise manner using Horner's rules.

Full-path signature

The computation of the *full-path* signature requires the state and event information. Each state and event is assigned a value that is used to compute the signature. The signature is evaluated using the path polynomial at a given point x_0.

Using this signature, one can detect incorrect paths. The states and the events must be visible to the signature generator. It is well suited for developing and implementing self-checking programs. The polynomial associated with a path C is defined as follows:

$$P_C(x) = \sum_{i=0}^{n-1} (s_i x^{2(n-i)} + e_i x^{2(n-i)-1}) + s_n \tag{7}$$

where:

s_i is the state value, e_i is the event value, and n is the length of the path.

[5] Without loss of generality, we can assume the signature of S_0 to be equal to zero.

The signature function Φ is defined as $\Phi(C) = P_C(x_0)$. The choice of x_0 is arbitrary as long as it is different from 0 and 1 [6]. The details about the choice of x_0 are discuused in thgis paper. Later, we discuss the method to assign values to states and events such that parallel paths have the same signature.

The event (state) signature is computed using only the event (state) information. The signature function for the event signature is given in Eq(8) and the signature function for the state signature is given in Eq(9) The event signature cannot detect all illegal state paths, and the state signature cannot detect all illegal event paths.

Event Signature:

$$P_C(x) = \sum_{i=0}^{n-1} e_i x^{n-i-1} \tag{8}$$

State Signature:

$$P_C(x) = \sum_{i=0}^{n-1} s_i x^{n-i} + s_n \tag{9}$$

Probability of detecting illegal paths

Theorem 1 *Prob($\Phi(C_1)$=$\Phi(C_2)$ | $C_1 \neq C_2$) = $\frac{1}{|F|}$ where $|F|$ is the size of the working algebraic field, where the state and the events can take any value in the Galois Field F.*

The detailed proof of the theorem can be found in the appendix. This theorem shows that the probability of detecting a fault is independent of the length of the path assuming that the states and events can be assigned all the values in F. This is briefly discussed in the following paragraph.

Choice of x_0

The choice of x_0 can be arbitrary as long as it is different from 0 and 1. 0 cannot be chosen because it results in a trivial signature and 1 does not allow to detect out of sequence events/transitions. In this section, we discuss the effect of x_0 on the masking probability and provide some hints to choose x_0 in order to have a good masking probability.

When computing the signature of a state (event) path, the state (event) values do not cover all the elements of the working Galois field. Consequently, the masking probability of the signature function may be sub-optimal. In order to obtain a good cover of the Galois field, one has to choose x_0 as a primitive root of unity. A primitive root of unity is an element w of $GF(p)$ such that $w^{p-1} = 1$ and $\forall i < p-1; w^i \neq 1$. Choosing x_0 a primitive root of the unity makes the set of possible signatures equal to $GF(p)$ even if the values of states (events) is restricted to a singleton a. For example, if a is the values assigned to a state S_a then the signature of path $S_a S_a \cdots S_a$ of length n is:

$$\begin{aligned} \Phi(a \cdots a) &= \textstyle\sum_{i=0}^{n-1} a x_0^n \\ &= a\frac{x_0^n - 1}{x_0 - 1} \end{aligned} \tag{10}$$

[6] Primitive roots of unity in a GF seems to be a good choice for value of x_0. The choice of x_0 is discussed in detail in a latter section.

When x_0 is a primitive root of unity, the range of the path signature is $GF(p)$ (for $0 \leq n \leq p-1$). Thus, with only one state/event values the state path signature covers the entire Galois field.

In [18], it has been proved that the condition on the choice of x_0 is a general condition which is equivalent to the condition used in the choice of a MISR circuit that has a good masking probability [7]. The condition for the choice of MISR circuits is that the compaction polynomial of the circuit must be a primitive polynomial[8].

Computation of run-time signature

The signature can be generated at run-time in the following manner.

Let x_0 be a given value chosen from a Galois field, and $S_1E_1S_2E_2S_3E_3 \ldots E_{n-1}S_n$ be the full path. $s_1, s_2, s_3, \ldots s_n$ are the values of the states and $e_1, e_2, e_3, \ldots e_{n-1}$ the values of events. At run-time, the signature is computed after every event. Horner rules are used to compute the signature [21].

$$Signature(i) := (Signature(i-1) * x_0 + s_i) * x_0 + e_i \tag{11}$$

Where $Signature(0) = 0$ and $Signature(i)$ is the signature after the i^{th} event has occurred.

Each computation of the signature requires at most two "+" and "*" operations. This shows that the signature is step-wise computable.

Advantages

1. Simplicity: It is step-wise computable and requires only "+" and "*" operations at each step.
2. State, event, and full-path signatures can be computed using the same technique.
3. All correct paths leading to the same state have the same static signature. This reduces the number of comparisons to verify the correctness of the run-time path.
4. The state and event-assignment problem can be solved in an algebraic manner. This is elegant and simpler compared to heuristic methods used for MISR signatures used in [8].
5. The proposed signature function provides a theoretical basis for calculating the masking probability.

OPTIMIZING THE COMPARISON

In this section, we discuss some of the techniques used to simplify the task of fault detection. Some of the issues discussed include parallel path signatures and the states where the static signature is compared with the run-time signature.

[7] The MISR circuit used by Leveugle is a particular case of the polynomial signature function used in this paper.

Signatures of parallel paths

When many paths lead to the same state, they are called *parallel paths*. In the case of parallel paths, our aim is to reduce the number of possible signatures. Ideally, all parallel paths should result in the same signature. This reduces the complexity of signature verification.

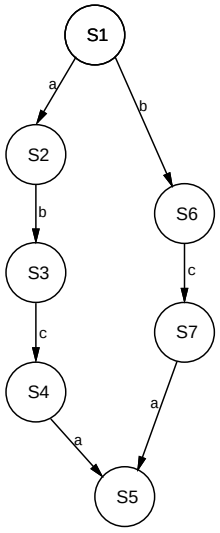

Figure 4 Parallel path.

For example, in *Figure 4* in the ideal case, the signature generated at the end of paths $S_1aS_2bS_3cS_4aS_5$ and $S_1bS_6cS_7aS_5$ are equal.

To solve this problem, we have to assign appropriate values to the events and states such that the signature of all parallel paths are equal. We denote this problem as the *event-state assignment* problem. When assigning values to the events and states, the following constraint has to be additionally satisfied: Two different events or states have to be associated with two distinct values; otherwise they are indistinguishable. The same signature is computed for both correct and wrong event. This makes it impossible to detect an incorrect transition resulting from a wrong event.

As the signature of a path is equal to the evaluation of its associated polynomial at a given point, solving the event-state assignment problem is equivalent to solving a system of linear equations.

In the case of full-path signature, the condition that two parallel paths must have the same signature is equivalent to solving a linear equation. In *Figure 4*, the linear equation is as follows:

$$s_1x^8+ax^7+s_2x^6+bx^5+s_3x^4+cx^3+s_4x^2+ax+s_5 = s_1x^6+bx^5+s_6x^4+cx^3+s_7x^2+ax+s_5$$

This equation can be simplified as follows:

$$s_1(x^8-x^6)+s_2x^6+s_3x^4+s_4x^2-s_6x^4-s_7x^2+ax^7=0 \tag{12}$$

If the signature is computed over the state path, then the equation is as follows:

$$s_1(x^4-x^3)+s_2x^3+s_3x^2+s_4x-s_6x^2-s_7x=0 \tag{13}$$

The advantage lies in the fact that only one signature is required for comparison with the run-time signature. This method does not require any adjustment values like other signature-based techniques [22]. It significantly reduces the complexity of the observing entity. It treats graphs with loops, in the same manner as graphs without loops. Finally the state or event values can be computed by solving a set of linear equations. Its disadvantage is the restriction on the topology of the control flow graph.

We now present a theorem that gives an idea about the solution to the event-assignment problem. The same result can be applied to the state-assignment and the event-state-assignment problem. In [8], provides the conditions under which the state assignment problem is solvable. We denote by $E^-(S)$ the set of edges that lead to state S and S^- the set of states S_i for which there exists an event e_i such that $\delta(S_i, e_i) = S$. S_0 is a state for which the signature is given an initial value (e.g., 0) and from which all other states are reachable.

Theorem 2 *The event-assignment problem is solvable if* $|E| > |E^-(S_0)| + \sum_{S \neq S_0}(|E^-(S)| - 1)$

The details of the proof of this theorem are provided in the appendix.

Figure 5 shows the fow graph corresponding to Algorithm 1. We now describe the algorithm for generating the system of equations to solve the *state-assignment* problem. The values of the states can be computed by solving the system of equations generated by this algorithm. The system of equations for computing the values of the events can be generated in a similar manner. The functions used in this algorithm have been defined in section on definitions and terminology. S^- denotes the set of states s_i for which there exists an event e_i such that $\delta(S_i, e_i) = S$.

We differentiate between the initial state S_0 and the other states. For each state in $S \in Q - \{S_0\}$, all paths starting from S_0 and leading to state S must have the same signature. In order to minimize the number of equations, we optimize the number of chosen paths leading to state S in the following manner. We first compute S^-, which is the set of predecessors of S. For every state S_i in S^-, we choose one path C such that the path starts at S_0 and leads to S and the penultimate state in C is S_i.

The state S_0 has a predefined signature equal to zero. Thus, for each state in S_0^- the algorithm generates a linear equation such that all the paths leading to S_0 have a null signature.

The proof of the algorithm can be done by induction on the number of states. This algorithm can be adapted to solve the event assignment problem and the state-event assignment problem [23].

Algorithm 1 *state-assignment problem(input: A, output: System)*

$System \leftarrow \{\}$
For all $S \in Q - \{S_0\}$ such that $|S^-| > 1$
  Let $S_1 \in S^-; C_0$ a state path such that: $first(C_0) = S_0; last(C_0) = S; penultimate(C_0) = S_1$
  For all $S_i \in S^- - \{S_1\}$;
    Let C_i a state path such that $first(C_i) = S_0; last(C_i) = S; penultimate(C_i) = S_i$
    $eqn \leftarrow \Phi(C_0) = \Phi(C_i)$
    $System \leftarrow System \cup \{eqn\}$
If $|S_0^-| > 0$ **then**

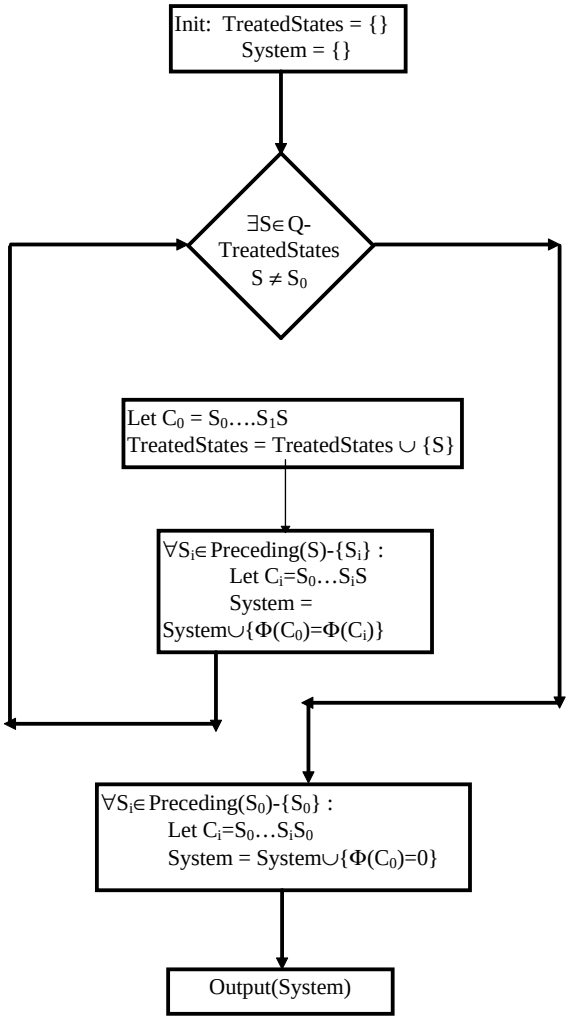

Figure 5 Flow chart for Algorithm 1

For all $S_i \in S_0^-$;

Let C_i a state path such that $first(C_i) = S_0; last(C_i) = S_0; penultimate(C_i) = S_i$

$eqn \leftarrow \Phi(C_i) = 0$

$System \leftarrow System \cup \{eqn\}$

end

This algorithm has a complexity of $O(|S|^3)$. The solution to the system of linear equations generated by the above algorithm can be solved using the Gauss-elimination method in $O(|S|^3)$. Theorem 2 discussed in the appendix gives the condition under which the state assignment problem can be solved. This result also holds good for the event assignment and the state-event assignment problems.

Checking States

The set of checking states $Q_c \subset Q$, is the set of states in which the static signature is compared with the run-time signature. When the program reaches a state $S_c \in Q_c$, the static signature associated with the state is compared with the run-time signature.

The size of the set Q_c determines the size of the static signature table. When Q_c is the same as Q, the comparison is made in every state. By choosing Q_c to be smaller than Q, it is possible to reduce the number of comparisons. The advantage of a small-sized Q_C is that the observer is simpler. However, this reduction is not for free. It increases the error latency[8]. A fault is not detected until the program reaches a state $S_c \in Q_c$. A reduced Q_c does not affect the probability of detecting a fault because the probability of detecting a fault is independent of the length of the path (refer to theorem 1). There is a trade off between error latency and the size of the static signature table.

This results in a dual problem. One may fix the size of Q_c and determine the elements of Q_c such that the error latency is minimized; or, for a given error latency, determine the elements of Q_c such the size of Q_c is minimized.

FAULT-DETECTION SCHEMES

In this section, we present the strategies for detecting run-time faults. A Static signature is computed for every state. This signature corresponds to the correct execution of the communication protocol. During the execution of the communication protocol, the run-time signature is computed for the messages exchanged between the communication entities. A fault is detected when the run-time signature does not match the static signature.

An observer is used to detect run-time faults. *Figure 2* shows the components of the observer. The observer consists of a *generator* and a *checker*. During the execution of the program, the run-time signature is computed by the generator. The technique described in the previous section is used to generate the run-time signature. The checker compares the run-time signature with the static signature stored in a table called the *static-signature table*. The comparison is performed when the program reaches the checking state.

Static-signature table

This table contains the signature corresponding to the correct execution of the communication protocol. The content of this table depends on the availability of the current state information for computing the run-time signature.

If the current state information is available to the generator, then this table has two columns: *checking state* and *signature*. This table contains the static signature for every checking state. If the current state information is not available, then this table has two columns represented by *previous signature* and *current signatures*. The signature of the previous comparison is used to find the static signature for the current comparison. After the comparison, the current signature

[8] In our case, the error latency is the number of events after which the fault is detected.

becomes the previous signature for the next comparison. In this case, the checker needs to know when the program reaches the checking state.

It is possible to reduce the number of comparisons by decreasing the size of the set of checking states Q_c. When the size of Q_c is less than the size of Q, the number of comparisons is reduced. This also reduces the number of rows in the static signature table. The advantage of this technique is the reduction of the size of the table. However, this increases the error latency.

In the following sections, we present methods for detecting faults. We discuss the structure of the static-signature table, requirements, advantages, and disadvantages of each scheme.

Fault detection without state information

In this case, the state information is not available to the generator. The event information is used to compute the run-time signature. The checker is notified when the program reaches the checking state. When the notification is received, the checker compares the run-time signature with the static signature for correctness.

Since the external checker does not have access to the state information, the table contains the previous signature and current signatures. The previous signature corresponds to the run-time signature when the last comparison was made. The comparison is made when the checker receives the notification from the program. After the comparison is made, the current signature becomes the previous signature for the next comparison.

The program must be modified to notify the checker when the program reaches the checking state. A single bit is sufficient for this notification. The generator and checker are independent of the program. The signature function and the static-signature table are the inputs to the observer.

Fault detection with state information

In this case, the generator has access to the current state information, the full-path signature is computed based on the state and event information. This method requires the program under observation to transmit the state information. It is necessary to extend the communication protocol to support this requirement.

The static-signature table has two columns: *current state* and *signature*. The static signature gives the signature for the current state and it is compared with the run-time signature. The size of the static-signature table can be reduced by reducing the size of the checking state.

With this approach the communication protocol must be extended to transmit the current state information. Every message exchanged between the entities must contain the current state of the program. The signature function, the static-signature table and the checking states are the inputs to the observer.

Type of faults detected

This technique detects faults when the run-time signature differs from the static signature. It can detect the following types of faults:

- **Illegal state/event path:** When the FSM mistakenly changes the state and traverses an illegal path, this results in an incorrect signature. The probability of detecting the illegal path is $1 - \frac{1}{|F|}$.
- **Out-of-order events:** When the events occur out-of-order, the run-time signature does not match with the static signature. These type of faults are detected using the full-path or the event signature.

Discussion

In this section, we compare the two techniques based on their advantages and disadvantages. For both the techniques, the size and complexity of the observer are smaller compared to those of the FSM-based methods. This argument is supported by the dimensions of the static signature table. The FSM table corresponding to the program has two dimensions (state and event). The current state and event are used to compute the next state. The static-signature table has only one dimension (state). For the given state (previous signature), it gives the correct signature. The observer uses only the current state or the previous signature to determine the static signature. This reduces the complexity of the observer by one dimension. Moreover, the number of entries in the static-signature table can be minimized by reducing the number of checking states.

The fault-detection mechanism is independent of the program under observation. The static-signature table and the signature function depend on the communication protocol. They are the inputs to the observer. Therefore, the observer can be used to detect run-time faults in any communication protocol. Moreover, the observer is not affected by the perturbations of the program's environment.

The overhead of transmitting the state information to the fault-detection module is greater than the overhead of transmitting the signal to perform the comparison. A fault due to an incorrect path may not be detected in the absence of the state information.

Once a fault has been detected, it has to be isolated and procedures have to be initiated to recover from the fault. After the recovery, the protocol may reset to initial state (like in TP4) or may continue from the last correct state depending on the nature of the fault. In the later case, it is necessary to analyze this scenario for the signature-based method. Can the signature-based method be used to roll back to the last correct state ? The answer to this question is yes (in the case when the observer has the state information) because it should be noted that the signature is calculated using the signature of the current state as the starting value. Therefore it is easy to roll back to the last correct state of the protocol and use the signature of that state as the starting value.

EXAMPLE

In this section, we take the FSM shown in *Figure 6* and analyze the signature generation method. We solve the state-event assignment problem and show the computation of full-path signature.

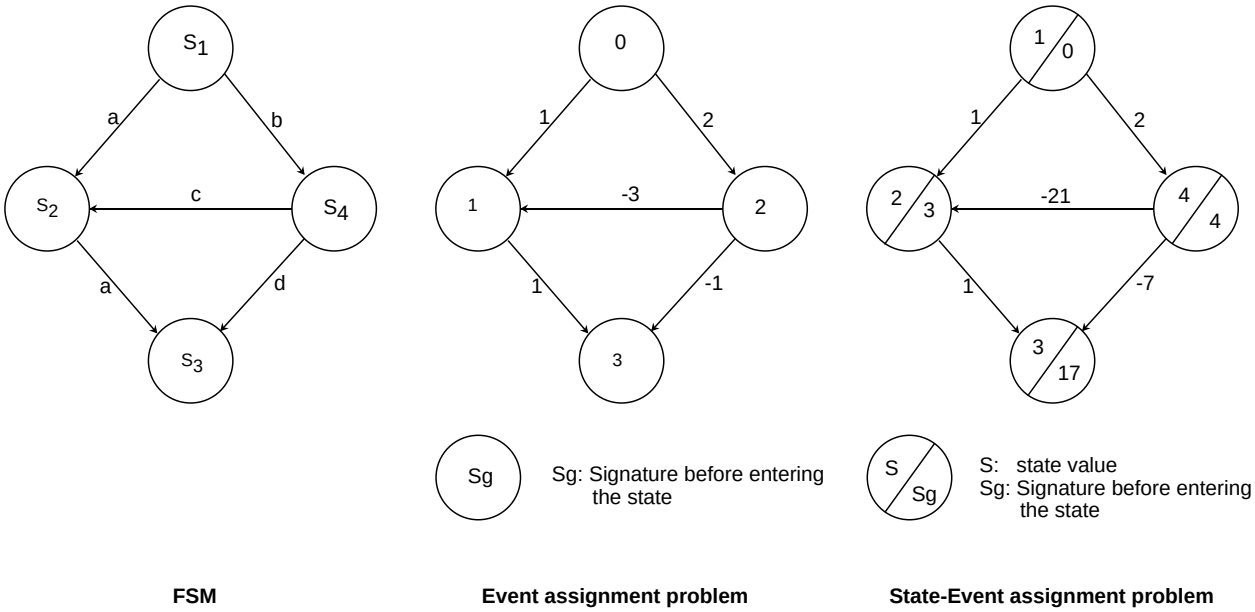

Figure 6 Event/State assignment example.

Event Assignment

Using algorithm 1, the event assignment problem can be transformed into the following equations.

$$\begin{cases} \Phi(a) &= \Phi(bc) \\ \Phi(aa) &= \Phi(bd) \end{cases} \tag{14}$$

Solving the event assignment problem is equivalent to solving the following system of equations:

$$\begin{cases} a &= bx + c \\ ax + a &= bx + d \end{cases} \tag{15}$$

The values of c and d can be deduced from the values of a and b.

$$\begin{cases} c &= a - bx \\ d &= a(x+1) - bx \end{cases} \tag{16}$$

If we take $x = 2$, $a = 1$, $b = 2$, then $c = -3$ and $d = -1$.

State-Event assignment problem

Using algorithm 1, the state-event assignment problem can be transformed into the following system of equations:

$$\begin{cases} S_1 x + a &= S_1 x^3 + bx^2 + S_4 x + e \\ S_1 x^3 + ax^2 + S_2 x + a &= S_1 x^3 + bx^2 + S_4 x + d \end{cases} \tag{17}$$

$$\begin{cases} c &= S_1(x - x^3) - S_4 x + a - bx^2 \\ d &= S_2 x - S_4 x + a(x^2 + 1) - bx^2 \end{cases} \tag{18}$$

For $x = 2$, $S_1 = 1$, $S_2 = 2$, $S_3 = 3$, $S_4 = 4$, $a = 1$ and $b = 2$, the value of (c, d) is $(-21, -7)$.

Fault detection

Table 1 Example1: Static Signature Table

State	Signature
S_1	0
S_2	3
S_3	17
S_4	4

We now describe the run-time signature generation technique for the FSM shown in *Figure 6*. *Table 1* shows the static signature table for this example.

Consider the following path: $S_1 b S_4 c S_2 a S_3$. *Table 2* shows the signature computed after each event. Both the event and state values are used in computing the signature. The signature computed after each event corresponds to the static signature of the state after the event (see *Table 1*). This means that the chosen path is correct.

Now let us consider an illegal path: $S_1 b S_4 c S_3$. *Table 3* shows the computation of the signature. After event c, the run-time signature is 3, which does not match with the static signature of S_3 (i.e., 17). So a fault is detected after event c.

Table 2 Example 1: Full-path Signature

Path	Computation	Signature
$S_1 b S_4$	$(0 * 2 + 1) * 2 + 2$	4
$S_1 b S_4 c S_2$	$((4 * 2 + 4) * 2 + (-21))$	3
$S_1 b S_4 c S_2 a S_3$	$((3 * 2 + 2) * 2 + 1)$	17

Table 3 Example 1: Incorrect Path

Path	Computation	Signature
$S_1 b S_4$	$(0 * 2 + 1) * 2 + 2$	4
$S_1 b S_4 c S_3$	$((4 * 2 + 4) * 2 + (-21))$	$3 \neq 17$

As mentioned in the section on the choice of the signature function, our method for detecting faults is probabilistic and Theorem 1 gives the upper bound on the probability of faults that can go undetected. An example of an undetected fault is as follows. Assume that there exists an event e that is assigned the value 3. If the event e is received when the current state is S_1 and the FSM goes from state S_1 to state S_3. This transition (S_1, e, S_3) is incorrect but cannot be detected by comparing the static signature 3 of state S_3 to the run-time event signature path $S_1 e S_3 = 0 \times 2 + 3$.

CASE STUDY: TP4

In this section, we explain the technique for generating the signature and detecting faults for the TP4 protocol. The FSM model of TP4 is shown in *Figure 7*. The visible events are *CR, CC, AK, DR, DC, DT, CR', CC', and AK'*. The input events and output events are distinguished

with a '. For sake of clarity, the tuple *(CR, CC, AK, DR, DC, DT, CR', CC', AK')* is denoted by *(a,b,c,d,e,f,a',b',c')*. Using algorithm 1, the event-assignment problem is transformed into the following the system of equations:

$$\begin{cases} \Phi(ab'c) &= \Phi(a'bc') \\ \Phi(ab'cc) &= \Phi(ab'c) \\ \Phi(ab'cc) &= \Phi(ab'cf) \\ \Phi(ab'cde) &= 0 \end{cases} \tag{19}$$

The third equation in system (19) gives the condition that events c and f must have the same value.

The other equations of system (19) can be transformed into a system of linear equations: $A[abcdea'b'c']^t = [00000000]^t$, where matrix A is defined as follows:

$$A = \begin{bmatrix} x^2 & -x & 1 & 0 & 0 & -x^2 & x & -1 \\ x^4 & 0 & x^2 & x & 1 & 0 & x^3 & 0 \\ x^2 - x & 0 & 1 & 0 & 0 & 0 & x-1 & 1 \end{bmatrix} \tag{20}$$

Solving this system gives a solution: $[c'ec]^t = B[abda'b']^t$, where B is given by the following equation:

$$B = \begin{bmatrix} x & -x & 0 & -x^2 & 1 \\ -x^3 & 0 & -x & 0 & -x^2 \\ x - x^2 & 0 & 0 & 0 & 1-x \end{bmatrix} \tag{21}$$

If we assign the value 2 to x_0 and $(a, b, d, a', b') = (1, 2, 3, 4, 5)$, then the tuple (c', e, c) has to be equal to $(-13, -34, -7)$. The solution is as follows: $(a, b, c, d, e, a', b', c') = (1, 2, -7, 3, -34, 4, 5, -13)$. *Figure 8*(a) shows the static signature for every state. *Table 4* gives the static-signature table corresponding to this FSM. This signature is computed using the event information.

Table 4 Static-Signature Table for 5(a).

Previous Signature	Current Signature
0	4, 1
1	7
4	10
7	7, 17
10	7
17	0

Figure 9 shows the reduced FSM with the checking states, $Q_c = (Closed, Open)$. *Table 5* gives the static signature corresponding to this FSM.

Table 5 Reduced FSM Static-Signature Table

Previous Signature	Current Signature
0	7
7	0, 7

The event-state assignment can be solved in a similar manner. A possible solution is shown in *Figure 8*(b). This solution is *(Closed, WFCC, WFCC', Open, WFTRESP, AKWAIT, Closing) = (0, 0, 0, 0, 0, 2, 0)* and *(CR, CC, AK, DR, DC, DT, CR', CC', AK') = (1, 2, -30, 3, -172, -30, 4, 5, -62)*.

More details on solving the event-state-assignment problem are discussed in [23, 24].

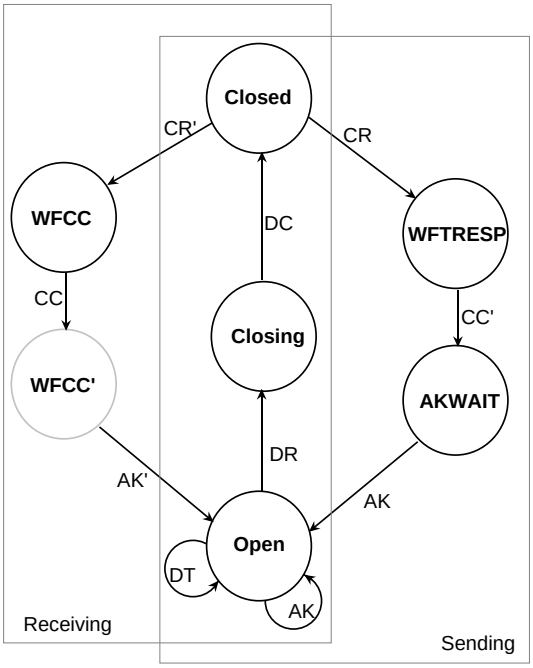

Figure 7 TP4: FSM representation

Fault detection

Table 6 Example 1: Static-Signature Table

State	Signature
$Closed$	0
$WFCC$	4
$WFCC'$	18
$Open$	10
$WFTRESP$	1
$AKWAIT$	9
$Closing$	43

We now describe the technique for generating the run-time signature for the TP4 protocol whose FSM is shown in *Figure 7*. This FSM corresponds to the correct execution of TP4. *Table 6* shows the table of static signatures for this example.

Consider the following correct path: (Closed CR WFTRESP CC' AKWAIT AK Open). This corresponds to the connection-establishment phase of TP4. *Table 7* shows the signature computed after each event. Both the event and state values are used in computing the signature. The value of x_0 is 2. The technique described in section on signature generation is used to compute the signature. The signature computed after each event corresponds to the static signature

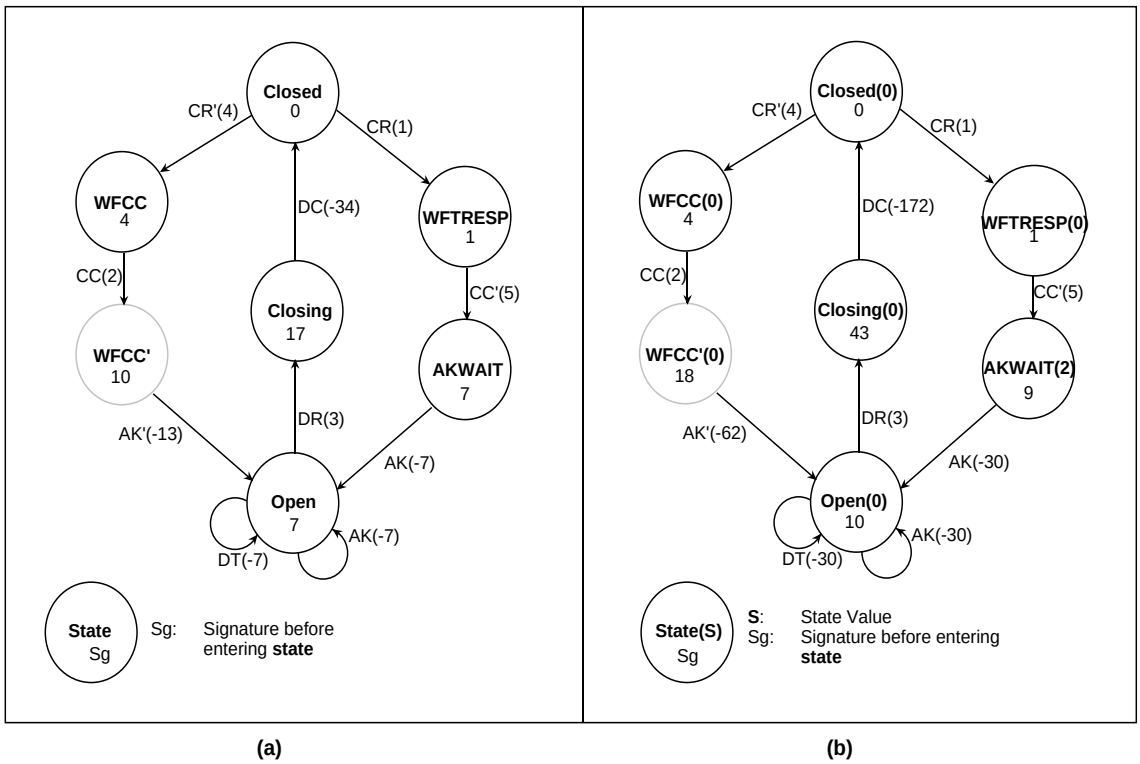

Figure 8(a) TP4 Protocol: event assignment solution **(b)** TP4: Protocol state and event-assignment

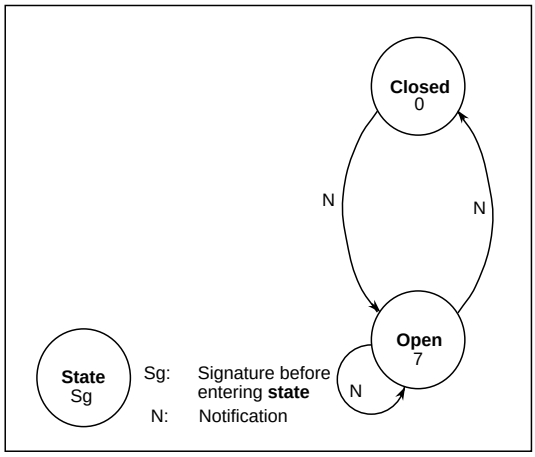

Figure 9 TP4 Protocol: Reduced FSM with checking states

in *Table 6*. This means that the chosen path is correct.

Now, let us consider an incorrect path: (Closed CR WFTRESP DR Closing DC Closed). *Table 8* shows the computation of the signature. After event DR, the run-time signature computes to 7, which does not match with the static signature of `Closing` (i.e., 43). Thus, a fault is detected after event DR.

Table 7 Example 2: Correct Path

Path	Computation	Signature
Closed CR	$0 * 2 + 1$	*1*
Closed CR WFTRESP CC'	$(1 * 2 + 0) * 2 + 5$	*9*
Closed CR WFTRESP CC' AKWAIT AK	$(9 * 2 + 2) * 2 + (-30)$	*10*

Table 8 Example 3: Incorrect Path

Path	Computation	Signature
Closed CR	$0 * 2 + 1$	*1*
Closed CR WFTRESP DR	$(1 * 2 + 0) * 2 + 3$	$7 \neq 43$

AN EXTENDED TRANSPORT COMMUNICATION PROTOCOL

TP4 transport protocol is designed to allow the transmission of additional parameters without deviating from the ISO standard [25]. In this section, we take advantage of this feature and we describe the extensions to TP4 required to support the computation and comparison of run-time signatures. The extensions do not modify the communication protocol but require the addition of some parameters to transmit the current state information. The original communication protocol can still be used to communicate with entities that use the extended communication protocol. Two extensions are proposed in this section.

- **Extension 1**: The current state of the program is included in every message exchanged between the communication entities. Every state of the program is assigned an integer value called the *state value*. The state value is used to compute the run-time signature.
- **Extension 2**: When the program reaches the checking state, a notification is included in the message. A single bit is sufficient to include this information. When this bit is set, the checker compares the run-time signature with the static signature for correctness.

Transport Class 4 (TP4) [25, 21] is used as an example to illustrate the two extensions. The proposed extensions to TP4 do not affect its functionality. The eXtended TP4 (eXTP4) can communicate with an existing TP4 implementation without any modifications to the TP4 implementation.

A Transport Protocol Data Unit (TPDU) has a fixed header, a variable header, and data (see *Figure 10*). The fixed header contains the mandatory parameters. The variable header contains the optional parameters. Each optional parameter in the header is encoded in the following manner:

parameter = (code, length, value)

The *code* represents the type of the parameter and the *length* indicates the number of bytes used to represent the *value*.

The optional parameters in the TP4 header facilitates the inclusion of the *current state* value in every message. The state information is treated as an optional parameter. It is ignored by TP4 and can be used by eXTP4 to compute the run-time signature.

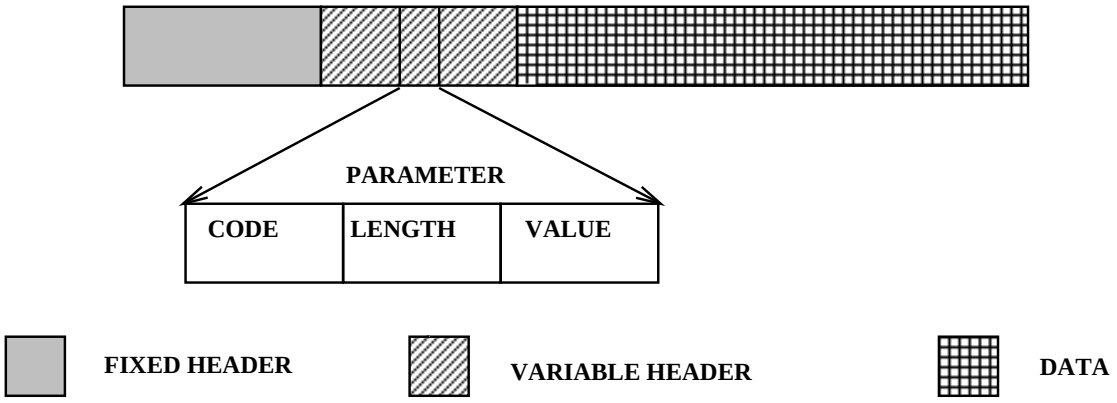

Figure 11: Transport Protocol Data Unit

For TP4, the value of $|Q|$ is nine. This requires one byte to represent the state information[9]. Three additional bytes are required to include the current state information in every message.

Extension 2 requires only a single bit to be included in the message when the program reaches the checking state. Since the minimum length of the value of a parameter is one byte, three additional bytes are required to include this information.

The code for the new parameters (current state and checking state) must be chosen such that there is no conflict with the code of existing TPDU parameters.

Overhead due to extensions

Table 9 shows the length of headers of different Transport Protocol Data Units. The overhead of both the extensions is three bytes and the overhead percentage is calculated as a percentage of the header length. It does not take into account the user data that is of variable length and is also a part of the TPDU. In many cases, the size of the data field is large compared to the size of the header. In this case, the overhead is negligible when computed for the overall frame length. The fifth and the sixth entries in Table 9 indicate the overhead for DT TPDU of size 512 and 1024 bytes respectively. It can be seen that the overhead is negligible for TPDU of size 512 and 1024 bytes.

Discussion

The number of bits required to include the state information is $log_2|Q|$, where $|Q|$ represents the number of states of the communication protocol. Only one bit has to be included in the message when the program reaches the checking state. In general, the communication overhead for Extension 1 largely exceeds that for Extension 2. The overhead of Extension 2 is independent of $|Q|$. On the contrary, the overhead of Extension 1 depends on $|Q|$. However, this overhead permits the detection of a broader class of protocol faults. In the case of TP4, both extensions require the same number of bytes. Extension 1 includes the current state information in every

[9] Four bits are sufficient to represent 9 states. Since the minimum length of the value is one byte, we use only the first four bits of value to include the current state information.

PDU TYPE	Hdr length	Overhead(%)
CR	35	8.5
CC	35	8.5
DT (Normal)	8	37.5
DT (Extended)	11	27.3
DT (512 bytes)	9	0.6
DT (1024 bytes)	9	0.3
ED (Normal)	8	37.5
ED (Extended)	11	27.3
AK (Normal)	12	25.0
AK (Extended)	17	17.6
AKf(Normal)	23	13.0
AKf(Extended)	27	11.1
EA (Normal)	8	37.5
EA (Extended)	11	27.3
DR	10	30.0
DC	9	33.3

Table 9: *Overhead for transmitting the current state and checking state information*

message. Extension 2 does not include the checking state information in every message. Thus, Extension 2 has less overhead compared to Extension 1.

It is possible to reduce the overhead of each extension by modifying the protocol. The information about the current state can be encoded in $log_2|Q|$ bytes. The information for the checking state can be included in a message using a single bit. This results in a modified protocol which requires new encoders and decoders to exchange the messages. In this case, the modified protocol cannot inter-operate with the original protocol.

CONCLUSION AND FUTURE WORK

In this paper, we have combined two methods: signature-based and observer-based methods, for detecting run-time faults in communication protocols. Our observer is simple compared to existing FSM-based methods.

We have proposed a new method for generating signatures based on polynomials. This method is simple, step-wise computable, and requires at most two additions and one multiplication for each computation of the run-time signature. We have provided an algorithm that assigns values to the states and events such that the number of comparisons are reduced. This method generates unique signatures that can detect illegal transitions and out-of-order events. Hence, this method is more reliable than other existing methods, which cannot detect out-of-order events.

We have proposed extensions to communication protocols to facilitate the application of this method to detect run-time faults. However, this extension results in some overhead. *eXTP4* is an extension to a transport protocol TP4 (*eXTP4*) without modifying the original protocol. Our future work will concentrate on applying this new method to communication protocols that are modeled using extended FSMs [11, 24].

APPENDIX

Probability of Detecting a Fault

Theorem 1 *Prob($\Phi(C_1)$=$\Phi(C_2)$ | $C_1 \neq C_2$) = $\frac{1}{|F|}$ where $|F|$ is the size of the working algebraic field.*

Proof: The probability that two different paths have the same signature is equal to the probability that two different polynomials have the same value at a given point x_0. This is equal to the probability that a polynomial evaluates to a value sgn at x_0. For any given working field F, the size of the set of polynomials of degree n is given by $|F|^{n+1}$. By symmetry, the size of the set of polynomials that evaluates to sgn at point x_0 is $|F|^n$. Let P_{C2} be the polynomial corresponding to path C_2. Then the probability that the polynomial P_{C2} evaluates to sgn is $|F|^n/|F|^{n+1} = 1/|F|$. □

Event Assignment Problem

Theorem 2 *The event-assignment problem is solvable if* $|E| > |E^-(S_0)| + \sum_{S \neq S_0}(|E^-(S)| - 1)$

Sketch of Proof: The event-assignment problem is equivalent to solving a system of linear equations. We have $|E|$ unknowns and $|E^-(S_0)| + \sum_{S \neq S_0}(|E^-(S)| - 1)$ equations, since for each state S different from S_0 we have $|E^-(S)|$ different paths. These paths have the same signature, which is equivalent to $|E^-(S)| - 1$ equations. For S_0, since the signature of all the paths that leads to it is predefined, we have $|E^-(S_0)|$ equations. Since this system of linear equations has a second member equal to 0, and the number of equations is greater than the number of unknowns, we can find a large number of solutions. □

Acknowledgments

The authors are grateful to Dr. Berthe Choueiry for reading the paper and providing valuable comments. This paper was partially supported by the Swiss PTT Project F&E N°309.

REFERENCES

1 A. Bouloutas, G. W. Hart, and M. Schwartz. Simple Finite-State Fault Detectors for Communication Networks. *IEEE Transactions on Communications*, 40(3):477–479, March 1992.

2 C. Wang and M. Schwartz. Fault Detection with Multiple Observers. *IEEE Transactions on Networking*, 1(1):48–55, 1993.

3 M. Riese. Diagnosis of Communications Systems: Dealing with incompletness and uncertainty. In *IJCAI-93 International Joint Conference on AI*, pages 1480–1485, August 1993.

4 M. Diaz, G. Juanole, and J. Courtiat. Observer- A concept for Formal On-Line Validation of Distributed Systems. *IEEE Transactions on Software Eng.*, 20(12):900–912, 1994.

5 S. S. Yau and Fu-Chung Chen. An Approach to Concurrent Control Flow Checking. *IEEE Transactions on Software Engg.*, 6(2):126–137, March 1980.

6 N. R. Saxena and E. J. McCluskey. Control-Flow Checking Using Watchdog Assists and Extended-Precision Checksums. In *The Nineteenth International Symposium on Fault-Tolerant Computing*, volume 19, pages 428–435. FTCS, IEEE Computer Society Press, June 1989.

7 S. J. Upadhyaya and B. Ramamurthy. Concurrent Process Monitoring with No Reference Signatures. *IEEE Transactions on Computers*, 43(4):475–480, April 1994.

8 R. Leveugle. *Analyse de Signature et Test en Ligne Integré sur Silicium.* PhD thesis, Institut National Polytechnique de Grenoble, 1990.

9 K. N. Oikonomou. Abstractions of Finite State Machines Optimal with respect to Single Undetectable Output Faults. *IEEE Transactions on Computers*, C-36(2):185–200, February 1988.

10 K. N. Oikonomou. Abstractions of Finite State Machines and Immediately-Detectable Output Faults. *IEEE Transactions on Computers*, 41(3):325–338, March 1992.

11 M. Riese. *Model-Based Diagnosis of Communication Protocols.* PhD thesis, Swiss federal Institute of Technology at Lausanne, Switzerland, 1993.

12 G. Noubir and K. Vijayananda. Robust Communication Protocols for run-time Fault Detection. In *ICCC'95, Twelfth International Conference on Computer Communication, August 20-25, Seoul, South Korea.* ICCC, 1995.

13 G. Noubir, K. Vijayananda, and P. Raja. Signature-based Fault Detection for Communication Protocols. In *ISIT'95 International Symposium on Information Theory, September 17-22, 1995, Whistler, B.C., Canada* . IEEE.

14 G. Noubir, K. Vijayananda, and H. J. Nussbaumer. A Robust Transport Protocol for run-time Fault Detection. In *ICNP'95 International Conference on Network Protocols, November 7-10, 1995, Tokyo, Japan.* IEEE.

15 K. Sabnani and A. Netravali. A High Speed Transport Protocol for Datagram/Virtual Circuit Networks. In *SIGCOMM'89,Symposium on Communications Architecture and Protocols*, 1989.

16 A. N. Netravali, W. D. Roome, and K. Sabnani. Implementation of a High Speed Transport Protocol. *IEEE Transactions on Communications*, 38(11):2010–2023, 1990.

17 B. T. Doshi, P. K. Johri, A. N. Netravali, and K. Sabnani. Flow Control Performance of a High Speed Protocol. *IEEE Transactions on Communications*, 41(5):707–719, 1993.

18 G. Noubir and B. Y. Choueiry. Algebraic Techniques for the Optimization of Control Flow Checking. In *FTCS26, The 26th Annual International Symposium on Fault-tolerant Computing, Sendai, Japan, June 25-27, 1996.* IEEE, 1995.

19 R. E. Blahut. *Fast Algorithms for Digital Signal Processing.* Addison Wesely, Reading, MA, 1990.

20 H. J. Nussbaumer. *Fast Fourier Transform and Convolution Algorithms.* Springer Series in Information Science. Springer-Verlag, 1982.

21 H. J. Nussbaumer. *Computer Communication Systems*, volume 1 & 2. Wiley Chichester, 1989.

22 K. D. Wilken and J. P. Shen. Embedded Signature Monitoring: Analysis and Technique. In *International Test Conference (ITC)*, pages 324–333, 1987.

23 G. Noubir. *Nouvelles Méthodes pour la Tolérance aux Pannes Basées sur l'algèbre des Polynômes.* PhD thesis, Swiss Federal Institute of Technology at Lausanne, Switzerland, 1996.

24 M. Riese. *A Framework for Diagnosis of Communication Protocols.* PhD thesis, Swiss federal Institute of Technology at Lausanne, Switzerland, 1993.

25 International Telecommunication Union. Transport Protocol Specification for Open System Interconnection for CCITT Applications. Ref. Number CCITT X.224, ISO 8073, Geneva, Switzerland, 1988.