A Survivable Critical Infrastructure Control Application*

A. Serageldin, A. Krings
Computer Science Department
University of Idaho
Moscow, ID 83844-1010

A. Abdel-Rahim
Civil Engineering
University of Idaho
Moscow, ID 83844-1022

ABSTRACT

A microcontroller-based system is described that utilizes real-time distributed sensor data to adapt a critical infrastructure control application to reflect current environment conditions. The system monitors the behavior of its software execution based on real-time analysis of certified frequency spectra and call-graph transition validations in order to detect and react to uncertified behavior and state transitions. The real-time data used to adapt the application is geographically distributed and redundant. The overall system is outlined with focus on the contingency management of the system. Finally, a basic reliability analysis is given as a tool to evaluate the impact of fail rate assumptions.

Keywords: Design for Survivability, Dynamic Monitoring, Real-time Profiling, Certification, Contingency Management, Critical Infrastructure

1. INTRODUCTION AND BACKGROUND

Most of today's critical infrastructures are controlled by computers and embedded systems that utilize networks during operation. Examples are embedded systems that control the electrical power grid, traffic controllers as part of intelligent transportation systems, or sensor systems used in weather prediction and reporting. Unlike legacy systems that operate independently as an isolated stand-alone system, today's control systems are guided by real-time data that helps adapt the application, e.g., update schedules to reflect expected power usage, change phases in traffic signals, or weather-induced changes to the operation of the system. This need for flexibility of control exposes the system to a wide range of security and Cyber threats in addition to those issues addressed by fault-tolerant design strategies.

Given the criticality of the systems, fault-tolerance and survivability considerations have to be designed into the system, rather than in an add-on fashion. Fault-tolerant design considerations include basic concepts like redundancy management, a contingency management system, and a good understanding of the issues that can affect the reliability, e.g., using reliability analysis. Survivability considerations are more difficult to incorporate since the faults that are to be considered include all security issues, including pathological cases, as they may result from insider attacks. The general expectation is that the system can "survive" different faults and continue to provide essential services [4].

*This research has been supported by grant DTFH61-10-P-00123 from the Federal Highway Administration - US DoT.

1.1 System Overview

The system that is the basis for this research is representative of any control application found in a distributed critical infrastructure, e.g., electrical power grid; however, in this case it turns out to be part of the intelligent transportation system. The application is an embedded system that takes real-time data from a national sensor network via the Clarus [1] database server over the Internet and incorporates this data in algorithms that control the behavior of traffic signal operation [6]. The basic system is depicted in Figure 1, which shows the application controller (in this case the traffic control system), the embedded system that receives environment data from any specified database server (e.g., Clarus server and local mirror site) over the Internet that is accessible via a local switch through a setup of routers and firewalls. The embedded system receives environment data at regular intervals and computes changes for the application controller, which it accesses though the local switch. The application controller can also function in an autonomous fashion, i.e., the embedded system only supplies additional functionality, but it is not necessary for operation. The functionality that the embedded system adds to the control application is the flexibility to change the traffic signal timing in real-time to account for adverse road conditions, e.g., due to rain, ice, or snow. For example, if the road surface is slippery then the yellow time of the traffic signals is extended to increase safety.

This brings up an important issue related to safety: the actual application controller is NTCIP compliant, i.e., the controller is governed by National Transportation Communications for ITS Protocol NTCIP 1202. This means that the application controller is designed to work within a specified safe range of parameters. Only the system parts in the shaded areas are part of our consideration. All non-shaded components are already in place in most modern traffic signal intersections. Furthermore, the addition of the embedded control system does not require any modification of the existing (non-shaded) system components.

1.2 Threat Space and Fault Assumptions

As systems communicate over the Internet the exact threat

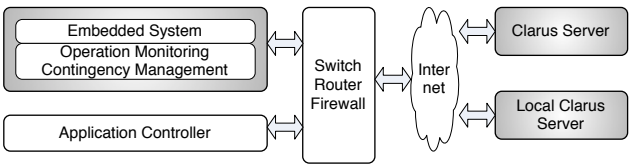

Figure 1: System Overview

space is unknown; however, it is fair to say that one inherits all security threats associated with Internet communication.

Our main concern is that the system operates within specifications and that any deviation from nominal behavior does not cause incorrect system states. Thus we need to consider any manipulation associated with access to the server, e.g., denial of service (DoS) attacks, code injection, buffer overflow, masquerading etc. The result of such fault scenario could be that the system (due to a software fault or attack) computes application parameters that reduce the effectiveness of the control application.

2. SYSTEM ARCHITECTURE

The system architecture combines traditional principles of fault-tolerance with the concept of *Design for Survivability* [4]. Whereas fault-tolerance addresses standard issues related to system reliability and safety, design for survivability is centered around a contingency management system that utilizes operation monitoring to detect and react to off-nominal executions to maintain essential functionalities. As mentioned above, the control application is augmented with the shaded subsystems shown in Figure 1.

At the core of the system is an embedded controller based on the RabbitCore RCM4300 Rabbit Microprocessor, which we will refer to as "Rabbit" in the following. The Rabbit uses *Dynamic C* version 10.70, which operates in a tasking model that strongly resembles non-preemptive scheduling. This version of Dynamic C supports Security Modules (AES and SSL), a FAT File System, Library Encryption Executable, the μC/OS-II Real-time Kernel supporting Point-to-Point Protocol, RabbitWeb and Simple Network Management Protocol (SNMP).

Dynamic C implements the non-preemptive tasking model using so-called *costatements*, which define the basic execution entities, similar to a non-preemptive process. The software system consists of several costatements (also referred to as costates), which execute in an endless loop, passing control from one costate to the next. The context switch from one costate to another is initiated by the currently executing costate itself by "yielding" calls. Thus this constitutes a dispatching environment that is based on "good behavior". Failing to yield would result in the current costate to monopolize the system. This can be safeguarded against by timer mechanisms, e.g., watchdog timers, which are also the reason why the execution model is not strictly non-preemptive in the scheduling theoretical point of view (which would not allow such preemption).

The Rabbit executes several costates, including the Application Control, which receives data, computes parameters and initiates changes to the application controller, and the Operation Monitoring and Contingency Management System, which will be described below.

2.1 Operation Monitoring

The software running on the embedded system shown in Figure 1 monitors itself to detect executions that are considered off-nominal (described below) using several mechanisms that can be mainly partitioned into *detection* and *recovery*. The main detection mechanisms are (1) frequency spectra analysis with detection of off-nominal patterns, (2) call-graph validation, which will be addressed in Section 2.3, in addition to (3) usual program exception handling. Recovery is initiated by the contingency management system, which takes advantage of data redundancy and extensive exception handling.

2.2 Off-nominal Executions

The software system is instrumented, allowing to generate frequency spectra, and thus profiles, of the execution system. It is based on the work in [3] and described in [5, 6]. If the observed profile deviates a certain distances from the expected profile then the profile is considered off-nominal. Only executions with nominal profiles are allowed to execute on the system. Any off-nominal execution indicates that such execution profile has not been observed before. This could be due to some unexpected changes of the environment, a software fault, an attack etc., in short: a so-far unobserved behavior. This approach was taken using simple threshold functions in [7] which were later refined on a costate-basis in [5]. Here we use the dual-bound and costate-based threshold functions described in [6], i.e.,

$$\epsilon^{max}[\alpha] = (\epsilon_1^{max}[\alpha], ..., \epsilon_{|M|}^{max}[\alpha]) \quad (1)$$

$$\epsilon^{min}[\alpha] = (\epsilon_1^{min}[\alpha], ..., \epsilon_{|M|}^{min}[\alpha]) \quad (2)$$

where $\epsilon_i^{max}[\alpha]$ and $\epsilon_i^{min}[\alpha]$ are the upper and lower threshold values of software module m_i in the set of modules M for costate α. Every observed profile over k epochs, denoted by $\hat{\mathbf{p}}^k[\alpha]$, that is in the region between the two vectors is assumed nominal. An *epoch* is defined between transitions from one operation to another. Thus we certify a profile $\hat{\mathbf{p}}^k[\alpha]$ to be a *nominal profile* if

$$\epsilon^{min}[\alpha] \leq \hat{\mathbf{p}}^k[\alpha] \leq \epsilon^{max}[\alpha] \quad (3)$$

Any off-nominal execution triggers an action from the contingency management system.

2.3 System States

As the system executes, the main operations, implemented by Dynamic C modules, form a state machine represented by the state machine shown in Figure 2. A total of 25 system states can be observed, i.e., $S_0, ...S_{24}$. For example, state S_0 indicates that the software is in function *main*(). There are two types of transitions between two states S_i and S_j, represented by arcs between states. The solidly drawn arcs represent *calls*, whereas dotted arcs represent *returns*, e.g., a call will cause the transition from S_1 to S_2, whereas a return will cause the transition from S_2 back to S_1. The state machine is derived directly from the software call graph during compile time of the program.

The system's state machine is directly related to the profiles that are derived during execution of the software. Specifically, the observed profile reflects the probability distribution of the system states. The system state machine is collectively exhaustive and states are mutually exclusive. Due to the fact that the system is a single processor, there cannot be true parallelism that could violate the assumptions

that at any given time the system is in only one state. Furthermore, during compile time a Dynamic C mapping file supplies the system call graph that is the basis for the state machine. The observed profiles are the basis for determining the threshold functions given in Equations (1), (2) and (3). In fact, the threshold functions can be determined using the approach discussed in [9], where execution sequences were used to determine the Markov chain and the state probabilities. In our case the Markov chain is derived not from execution sequences, but during compile time. The state space of this Markov chain should not be confused with the state space reflecting any possible execution sequence, which could be potentially very large. Our state space is at most the size of the vertex set of the call graph. Any violation of execution transitions, thus causing an illegal transition, is captured.

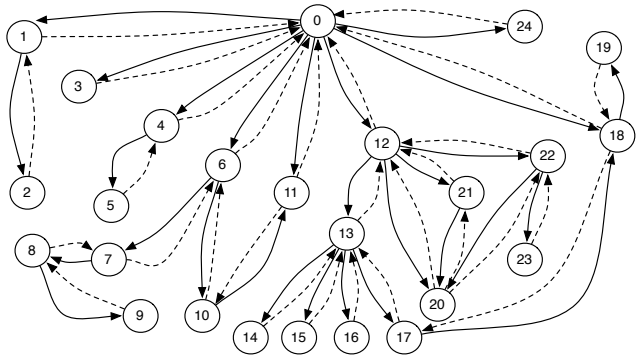

Figure 2: System State Machine

Any violation of transitions during system execution is detected. Specifically, detected are (1) the transition to an incorrect state as the result of a *call*, e.g., due to a code injection attack and (2) the return to an incorrect state as the result of a *return*, e.g., as the result of a buffer overflow attack. Any violation of state transitions is detected and activates the contingency management system.

3. CONTINGENCY MANAGEMENT

The Contingency Management System (CMS) is that part of the system managing decisions that need to be made as the result of faults or any external or internal events that should initiate some reaction, e.g., recovery to a certain safe state.

3.1 Application Control

The sub-system for the application control, i.e., the costatement that implements the real-time traffic control application, is shown in the simplified flow chart in Figure 3.

Since the real-time condition data is available from the Clarus server for specific subscriptions (indicated by subscription numbers) during fixed time intervals, e.g., every 15 minutes, the Rabbit determines the time and composes the URL that contains the comma separated values (a csv file). It then uses a recovery block strategy [8] to get the data from a set of data base servers. In our current implementation this is a Local Clarus Server (LCS) and the actual Clarus server, thus implementing a dual redundant system. First the LCS is queried and if the data cannot be retrieved within a certain amount of retries, then the Clarus server is tried. If it fails to provide the data after a certain amount of retries, the CMS initials *fail-safe mode*, which is a forward recovery mechanism bringing the system into a desired default state. Once entering fail-safe mode the system attempts to establish normal operations again. If data acquisition was successful via one of the alternatives, then the traffic controller adjustments that reflect the environment parameters are computed and the signal controller is adjusted correspondingly.

The data supplied by the Clarus server is assumed to be correct. This assumption is not unrealistic, since Clarus is designed using quality checking algorithms [2]. Thus the input data to the application control is assumed correct. However, the computed signal adjustment values, as computed by the Rabbit, are not assumed to be correct, as a fault may have occurred during computation. Therefore a Validity Check is implemented that tests the computed signal changes for range violations. If such violation is detected the CMS enters fail-safe mode. There are many other checking mechanisms implied in the flowchart of Figure 3, including reaction to network connection problems, data corruption, loss of time synchronization, e.g., after reboot as the result of power failure, inability of finding valid Clarus data subscriptions, changing the LCS Internet address, etc. Some of these issues require the CMS to enter a *receive mode*, in which configuration information, e.g., the IP of a new LCS or Clarus subscription number, is communicated to the system.

3.2 Monitoring Executions

The monitoring costatement (which is not depicted in the flow chart of Figure 3) implements system software monitoring and analysis of the observed profiles to determine if the execution is nominal, i.e., it conforms to Equation (3). If an off-nominal execution is detected, i.e., Equation (3) is violated, the system enters fail-safe mode. Thus, no off-nominal execution is allowed to make adjustments to the traffic controller. The reason for having an off-nominal execution may not have to be malicious in nature nor does it imply a fault. It may simply reflect an operation scenario that was not observed previously, e.g., at the time of system tuning before deployment. The CMS can send messages to that extend, thus allowing for an analysis of the scenario, which in turn may cause an updating of the threshold functions in Equations (1) and (2).

Any violation of the system state machine of Section 2.3 indicates a serious problem, as it implies that the system is calling modules that it should not be calling or it is returning to modules other than intended, e.g., as the result of a buffer overflow. The CMS initiates fail-safe mode and issues a notification about the nature of the violation.

As the result of monitoring the execution of off-nominal behavior and state transition violations, the system operates in a rather inflexible fashion. Whereas this could be undesirable in a generic computing application, most embedded systems have limited functionality and our system in particular is very deterministic in its execution due to the non-preemptive scheduling model.

3.3 Reliability Considerations

The reliability of the traffic control system is not affected by the embedded system since none of the components of the original signal control systems are modified. Recall that the embedded system implements only *added value*, but not

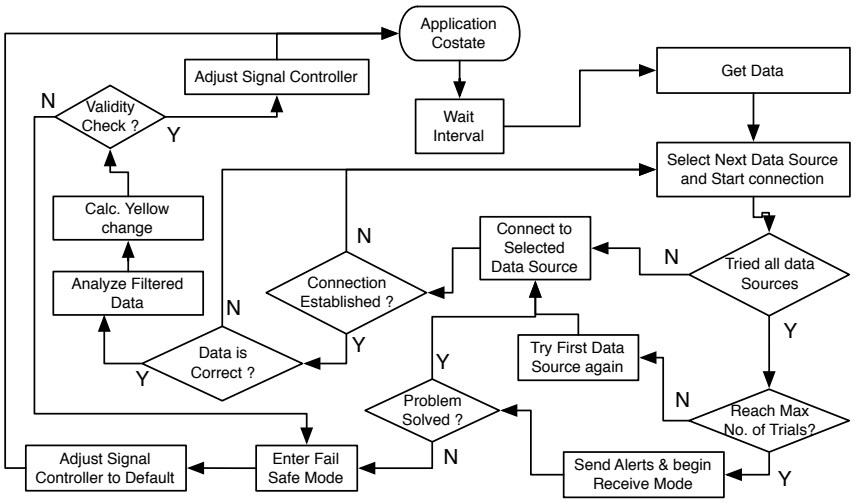

Figure 3: Flowchart of Application Control Costatement

basic functionality. In fact, as mentioned before, the traffic controller is NTCIP compliant. Thus action movie scenarios like an "all-green intersections" or "split second yellow timing" causing accidents are not possible (with or without the embedded system), e.g., attempts to assign parameters that violated NTCP compliance during testing of the embedded system were simply ignored by the traffic controller. The only physical connection with the existing infrastructure is via the connection to the switch.

The embedded system and its other components, as shown in Figure 1, can be modeled by the simple Fault Tree shown in Figure 4. The failure scenario is that the embedded system fails to provide added functionality. It should be noted that if the Clarus server (or the network to it) fails, then the LCS server is of no use anymore either, since it is only a mirror site, whereas if the LCS fails, the Clarus server can provide services.

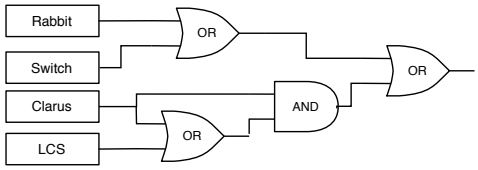

Figure 4: Simplified Fault Tree of System

4. CONCLUSIONS

The principles of fault-tolerant design and design for survivability were applied to a control application in a critical infrastructure. An embedded system was described that provided added value by incorporating real-time environment data to increase safety. The system implements a contingency management system that responds to the detection of off-nominal executions based on dual-bound observed profiles as well as detection of state transition violations. Thus the system observes any changes in its nominal execution, e.g., as the result of faults or malicious act, and any deviation from valid execution traces, e.g., as they may result from code injection or buffer overflow.

5. REFERENCES

[1] The Clarus System: http://www.clarus-system.com/

[2] RITA Intelligent Transportation Systems Joint Program Office, *Clarus Quality Checking Algorithm Documentation Report*, Final Report, December 21, 2010, FHWA-JPO-11-075.

[3] A. Krings, et al., *A Two-Layer Approach to Survivability of Networked Computing Systems*, Intl. Conf. on Advances in Infrastructure for Electronic Business, Science, and Education on the Internet,L'Aquila, Italy, Aug 06 - Aug 12, 2001.

[4] A. Krings, *Survivable Systems*, Chapter 5, Information Assurance: Dependability and Security in Networked Systems, Morgan Kaufmann Publishers, ISBN: 978-0-12-373566-9, 2008.

[5] A. Krings, V. Balogun, S. Alshomrani, A. Abdel-Rahim, and M. Dixon, *A Measurement-based Design and Evaluation Methodology for Embedded Control Systems*, Proc. 7th Annual Cyber Security and Information Intelligence Research Workshop, CSIIRW'11, Oct. 12 - 14, 2011, ORNL.

[6] A. Krings, A. Serageldin and A. Abdel-Rahim, *A Prototype for a Real-Time Weather Responsive System*, Proc. Intelligent Transportation Systems Conference, ITSC2012, Anchorage, Alaska, 16-19 September, 2012.

[7] John Munson, Axel Krings and Robert Hiromoto, *The Architecture of a Reliable Software Monitoring System for Embedded Software Systems*, ANS 2006 Winter Meeting and Nuclear Technology Expo, 10 pages, 2006.

[8] Brian Randell, and Jie Xu, *The evolution of the recovery block concept*, In Software Fault Tolerance, John Wiley & Sons, pp.1-22, 1994.

[9] Whittaker James A., and J.H. Poore, *Markov Analysis of Software Specifications*, ACM Transactions on Software Engineering and Methodology, Vol.2, No.1, January 1993, pp. 93-106.