Characterization of Error Resiliency of Power Grid Substation Devices

Kuan-Yu Tseng, Daniel Chen, Zbigniew Kalbarczyk, Ravishankar K. Iyer
Center for Reliable and High-Performance Computing
University of Illinois at Urbana-Champaign
1308 W. Main Street, Urbana, IL 61801, USA
{ktseng2, dchen8, kalbar, iyer}@illinois.edu

Abstract*— With the advent of modern technologies, computer-based devices are used to monitor and control critical infrastructures e.g., electric power grid, oil and gas distribution. However, the security and reliability of these computer-based systems becomes a significant issue since they are more susceptible to transient errors and malicious attacks. An error in one of these systems could have a cascading and catastrophic impact on the whole infrastructure. This paper explores the error resiliency of the power grid substation devices. Software-implemented fault injection technique is utilized to induce errors/faults inside devices used in the context of power grid substations. The goal is to test the systems' ability to compute through errors/faults. Our results demonstrated that a single error in substation device may cause the operator in the control center unable to control the operation of a relay in the substation.

Keywords: reliability, security, power grid; fault injection

I. INTRODUCTION

Computer-based control systems are becoming widely used in industrial processes for critical infrastructures such as transportation, waste water management, electricity delivery, and oil and gas distribution. As a result, the fault tolerance and error resiliency [1] of these computer-based control devices are also becoming a significant issue because of their sensitivity to transient errors[2] and malicious attacks.

Power Grid is a crucial infrastructure where a failure could have catastrophic impact on each citizen and on the society as a whole. To assure error resiliency, the power grid uses Supervisory Control and Data Acquisition System (SCADA) to monitor and control the health of critical assets such as generators, transmission lines and transformers. The growing dependency on computer-based protection e.g., use of Intelligent Electronic Devices (IED) in substations to monitor the power grid and communicate between substations and control centers makes this infrastructure vulnerable to transient errors and malicious attacks.

The lack of appropriately scrutinizing and maintaining system software in the control systems, and flawed design and implementation of critical support infrastructure are two of top 10 vulnerabilities of control systems according to the 2006 report of U.S. Department of Energy [1]. Furthermore, a 2007 survey by NERC [2] indicates that malicious attacks are a significant concern to power grid owners, operators, and regulators. Therefore, assessing the reliability and security of substation devices is necessary and urgent.

This paper characterizes the error sensitivity of the substation devices using black-box testing with the goal to improve the error resiliency of the devices. Software-implemented fault injection is used to investigate the complex interactions between faults/errors and error handling mechanisms inside target devices and to generate rare but severe failures which could happen in real scenarios.

As a case study, faults/errors are injected to the instructions and processor registers of the chosen application on the substation device. The chosen substation device in this case is a data aggregator, which plays an important role as a communication bridge between the control center and the substation in our example power grid infrastructure. The injected disturbances mimic the impact of transient errors and malicious attacks on applications executing on the devices. The key observations from this study are summarized as the following:

- Silent data corruption (SDC) can cause an operator in the *Control Center* to lose control over the equipment in the substation. Our result shows if a fault occurs in *dnpclient* and *dnpserver*[3], there is 13% and 7% chance that the fault will result in silent data corruption, i.e., the two applications apparently stay alive (e.g., can still accept commands from the *Control Center*) but do not operate correctly, e.g., cannot pass/execute commands and/or deliver current data from sensor devices in the substation. SDC consequences can be catastrophic to the system and the lost control over the substation may result in a cascading failure or damage of the electrical equipment.
- Error propagation (from application to the operating system) can lead to deadlocks of system resources. For instance, misbehaving *dnpclient* (e.g., cannot pass

[1] The resiliency here means the ability of the systems to sustain transient errors and malicious attacks.

[2] Transient errors (or accidental errors) mean the state of the system is changed due to physical environment, such as changing temperature or cosmic rays, unpredictably and temporarily.

[3] *dnpclient* and *dnpserver* are two critical applications executing on RTAC and responsible for (i) aggregation of data from substation sensors and (ii) passing commands from the control center to the relay in the substation. For more details see the discussion in Section V.

command to or acquire data from the relay) may lock system services, such as *kill* command. As a result, a forced reboot of the device is required to restart the offending application.

- Silent data corruptions in the monitoring software (the *monitor_app* in our study) can cause the system unable to recover the critical applications (i.e., *dnpclient* and *dnpserver*) from crash failures. Moreover, crash and subsequent restart (by operating system) of the monitoring software may cause the *monitor_app* to spawn new instances of critical applications even if there are already running instance.

This study shows that software-implemented fault injection provides a powerful method to conduct inexpensive black-box assessment of the error resiliency of devices in the power grid substation. The measurements allow us to: (i) obtain a good understanding of the interactions between system components (both physical devices and software), (ii) accurately trace error propagation and determine the failure cause, and (iii) provide feedback for the developers and vendors to refine the design and improve error resiliency.

II. Related Work

The interdependencies between the equipment inside power grid infrastructure make the vulnerability assessment complicated, as a failure in one device could lead to a cascading failure on others. A number of studies related to this paper have been carried out to evaluate the reliability and security of the power systems under this complexity.

Some are general studies such as the report from Idaho National Laboratory [6] which identifies many vulnerabilities in the energy delivery control systems. Other studies focus on the cascading failures of the power systems. For example, Mili and Qiu [7] assess the risk of catastrophic failures caused by a sequence of line tripping that results in voltage collapse. Anwar, et al. [8] put emphasis on the cyber security of the power grid and proposed a toolkit to automatically perform security assessment using the static and dynamic properties of power grid. The aforementioned studies, although aiming at different targets, all try to capture the interdependencies between grid components and analyze the power grid behavior under failures.

Fault injection is a useful technique to characterize the failure behaviors and evaluate the robustness of computing systems. It has been broadly studied and applied in many areas, including operating systems [3], cloud computers [4], and networks [5]. However, to the best of our knowledge, only few studies apply this technique on evaluating the reliability of the power grid. For example, Faza, et al. [9] analyze the reliability of the power grid by injecting faults into the FACTS devices (Flexible AC Transmission System), which are employed to adjust the flow of transmission lines. Nonetheless, they only inject software faults to the maximum flow algorithm used by the FACTS devices and give no consideration to transient errors and other critical devices in the substation. Rigole, et al. [12] discuss the resiliency of distributed microgrid control systems against network latency, node failures, and malicious attacks by utilizing fault injections. However, they don't take Silent Data Corruption (SDC) into account.

III. Testbed Setup

This study uses Software-Implemented Fault Injection (SWIFI) techniques to explore the error sensitivity of devices in the power grid substation. The analysis is conducted in an experimental testbed with real-world devices[4] and software to mimic realistic configuration and operation of the substation.

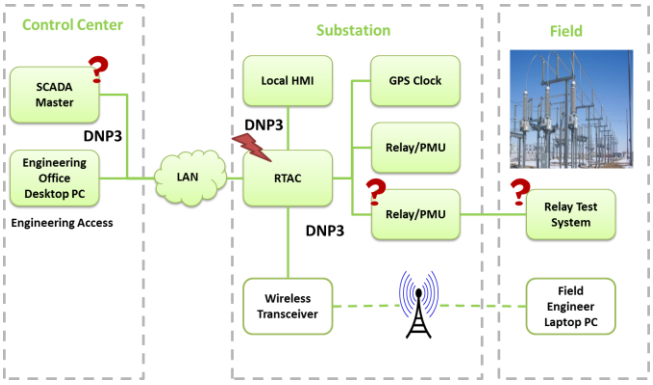

Figure 1: Testbed System Architecture

A. System Overview

Since it is infeasible to conduct experiments directly on real power infrastructure, the testbed is built to mimic power grid operations, including gathering data from sensors, device control and configuration, and response to emergency events. Figure 1 summarizes the configuration of the testbed, which is comprised of three major parts: *Control Center* (simulates operation of a control center), *Substation* (mimics a substation configuration and operation using real devices), and *Field* (simulates states of transmission lines providing inputs to the substation). The *Control Center* contains the SCADA Master, which collects data from *Substations*, analyzes the data, and issues commands to devices in the *Substation*. The SCADA Master in our testbed is simulated by a personal computer. The *Substation* is made up of different IEDs (Intelligent Electronic Device), responsible for monitoring local sensors, controlling local actuators, and communicating with the *Control Center* (see the next subsection for details on the *Substation* configuration). Due to the fact that it is hard to have a real power lines in the testbed, the *Field* consists of a simple simulator which simulates the value of currents and voltages of different transmission lines.

The *Control Center* is connected to the *Substation* through LAN (local area network), while the *Substation* is physically

[4] The model name of each device is removed for protection of the manufacturer.

(direct wires) connected with the *Field*. Note that except the field data simulator and SCADA software, all other devices in the testbed are real power equipment.

B. Substation

Since this study focuses on the resiliency assessment of a *Substation*, this section introduces the devices used in our testbed to mimic realistic *Substation* configuration.

Relay. The *Relay* used in this case study is a protective relay. It can monitor the health status of the transmission lines and can be configured for phase synchronism checks.

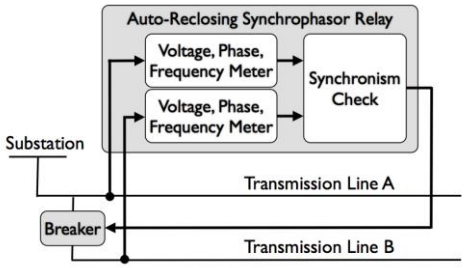

Figure 2: Auto-Reclosing Synchrophasor Relay Operation

Figure 2 illustrates the use of auto-reclosing synchrophasor *Relay* to actuate a circuit breaker connecting two parallel high-voltage transmission lines. The relay taps measurements from each of the lines and controls the state of the circuit breaker. In the case that the circuit breaker had to be closed, the two parallel lines must be synchronized before the breaker can connect them. The conditions for connecting two transmission lines are that the voltage, frequency and phase differences must be below a safety threshold, otherwise damage to the lines and/or power equipment can occur. The relay process these values on each of the parallel lines, calculates the differences, and compares the difference to safe boundaries, a process known as synchronism check.

Real time automation control (RTAC). *Teh RTAC serves as a data aggregator and a gateway. As a* data aggregator, it collects currents, voltages, and phasor data from each of the *Relays* for analysis and display in the local HMI (Human Machine Interface). As a gateway, it accepts connections from the SCADA Master in *Control Center*, responds to the requests, or passes commands to the designated *Relay*, allowing operators to acquire sensor data or control *Relays* from outside the *Substations*.

GPS Clock. This is a satellite-synchronized clock, which calculates the precise time from GPS signals with ±100ns average and ±500ns peak error. It is connected to *Relay* to provide accurate time stamp for the measurements collected by the *Relay*.

Local Human Machine Interface (HMI) This is an Embedded Automation Computing Platform serves as a HMI that provides the visualization of the power system. It retrieves sensor data from RTAC and displays them textually or graphically for the operators in *Substations* to monitor the condition of the whole *Substation.*

Wireless transceiver. This device provides encrypted wireless channels for the communication between two remote serial ports or a remote serial port and a field engineer's laptop PC. It is used for engineering access when physically connecting to electrical equipment is cost-ineffective or impractical.

C. Operational Scenario

The field value simulator simulates the states of transmission lines in the *Field*, from which each of the *Relays* in the *Substation* reads currents, voltages, and phasors of the transmission lines, timestamps them with the GPS Clock, and sends the data to the RTAC. RTAC aggregates data from *Relay*s and sends them to the SCADA Master in the *Control Center*. Upon receiving data, SCADA Master analyzes them to determine the health of the power grid and takes actions in response to anomalous events. For example, if the power grid is under stress and the operator in the *Control Center* wants to shed some load, the operator can issue a command to the *Substation* to open the breaker monitored by the corresponding *Relay* to prevent possible catastrophic failure. RTAC then passes this command to the *Relay*, which upon receiving the command, opens the breaker. All communications between SCADA Master, RTAC, and *Relay* use DNP3 protocol[6], which is widely employed in the industrial control systems including the power grid infrastructure in the North America.

IV. METHODOLOGY

Software-Implemented Fault Injection (SWIFI) is used to evaluate and characterize the behavior of power grid equipment in the presence of errors. A fault injector based on `ptrace()` feature in Linux OS is developed to automatically inject faults to instructions and registers, which can mimic the impact of transient errors and malicious attacks in the target device. The fault injector can be easily extended for NFTAPE injection framework [13]. Experiments are conducted on three important applications running on RTAC: *monitor_app*, *dnpclient*, and *dnpserver*. We focus on RTAC device because of its key role in: (i) aggregating data form other power equipment devices in the *Substation* and communicating them to the *Control Center* and (ii) receiving and passing commands from the *Control Center*, e.g., tell a *Relay* in the *Substation* to open/close breaker.

A. Target System Selection

Among all IEDs in the *Substation* (in our testbed settings), RTAC is the communication bridge between SCADA Master and other substation devices. All control commands

[6] The DNP3 protocol defines a client/server protocol stack that can run on top of serial ports or Internet Protocol (IP) supporting data acquisition and process control, e.g., using the devices in the substation.

sent to or sensor data received from *Relay*s pass through the RTAC. Consequently, transient errors in hardware can paralyze the communication between substation devices and the *Control Center*, which could lead to bad operator decision and cause further cascading failure. In addition, compromising the RTAC would allow attackers to control the breakers monitored by the *Relay*s or tamper with the sensor data so as to sabotage the electrical equipment or permanently damage the power grid infrastructure.

B. Fault Injection Environment

RTAC runs a customized real-time Linux on a single-core PowerPC CPU. **Error! Reference source not found.**summarizes the system configuration. Three important applications running on RTAC are targeted by our fault injection study:

1) monitor_app: responsible for monitoring and recovering (by restarting) other critical applications such as *dnpclient* and *dnpserver*.

2) dnpclient: responsible for the communication between RTAC and the remote DNP server running on another substation device such as *Relay* in our settings.

3) dnpserver: supporting communication between RTAC and the remote DNP client such as SCADA Master in our settings.

Error! Reference source not found. illustrates the working scenario between SCADA Master and *Relay* through *dnpserver* and *dnpclient*.

C. Approach

The fault injection process includes the following steps (as illustrated in **Error! Reference source not found.**).

STEP 1: Determine injection configuration. Before fault injections, three parameters are specified, *target application*, *MTBF(Mean Time Between Faults)*, and *injection count*.

- *Target application* specifies the name of the target application to be analyzed.
- *MTBF* specifies mean time between faults in terms of number of instructions.
- *Injection count* specifies how many faults in total will be injected into the target application.

STEP 2: Generate injection targets. The *Injection Target Generator* randomly selects: (i) the target location– an *instruction* or *a general purpose register* and (ii) the fault type – *Bitflip* (bit-flip), *Add* (adding a constant), *Swap* (byte swap), *Overwrite* (data overwrite) or *NOP* (replacing an instruction with no operation). The register injection can effectively mimic the faults occurring in the data segment because the PowerPC architecture always loads the data into the register before operating on the data.

STEP 3: Inject a fault. When the *Fault Injector* receives necessary information (i.e., *target application, MTBF, injection count, target location, and fault type)* from *Injection Configuration Manager* and *Injection Target Generator*, it uses `ptrace()` to attach to the target application for tracing the application execution. At that point the application process is suspended. Next, the fault injector steps over *N* instructions, where *N* is drawn from the Poisson distribution with the mean of *MTBF* (to mimic random arrival of errors). Next, the injector inserts a fault according to the predefined target location and the fault type. Finally, the injector decreases the *injection count* by one, resumes the application, and continue with the next injection if the *injection count* is greater than 0 (i.e., do STEP 3 again). Note that in this way, the injected faults are always activated, which means the error activation rate is 100%.

STEP 4: Validate the fault impact. For each application, an application-specific validation process is executed by the *Validator* to determine the impact of the injected fault. If the application crashes, it is restarted. If the fault causes the whole system to malfunction, the system is rebooted.

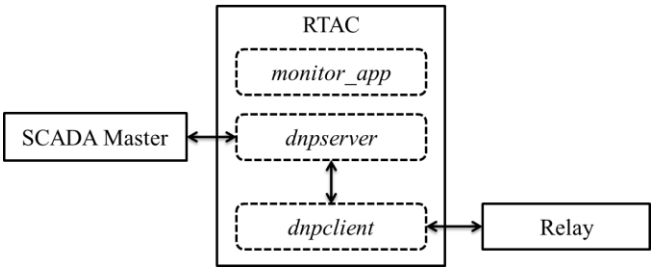

Figure 3: RTAC Working Scenario

TABLE I. RTAC SYSTEM CONFIGURATION

Processor	Memory	Storage	OS	Ethernet	Serial
PowerPC 533MHz	512MB DDR2 ECC	4GB (2GB Reserved)	Linux 2.6.29.3 RT PREEMPT	2 ports 10/100 Mbps	4 ports

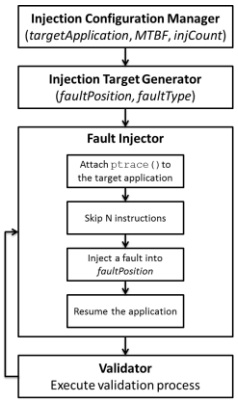

Figure 4: Fault Injection Process

D. Validation Process

The validation process for each application is the following:

1. ***monitor_app* –** the *Validator* terminates the monitored application (e.g., *dnpclient*) and if the application cannot be restarted by the *monitor_app*, the *monitor_app* is declared as failed.

2. ***dnpclient*** – *dnpclient* serves as the communication bridge between RTAC and *Relay*. Therefore, the *Validator* directly opens and closes the breaker in the *Relay* and makes the RTAC to pull sensor data to check whether the currents and voltages change according to the changes in the breaker position.
3. ***dnpserver*** – *dnpserver* is responsible for: (i) passing breaker control command from SCADA Master to *dnpclient,* which in turn passes it to the *Relay* and (ii) sending the sensor data acquired by *dnpclient* from *Relay* to the SCADA Master. Thus, the *Validator* issues "breaker open" command from the SCADA Master to the *Relay* via RTAC, and acquire currents and voltages from the RTAC to check whether the currents drop to zero due to the open breaker. Then, the *Validator* issues "breaker close" command and check whether the the currents change back to the nominal values. If one of the commands is not successful, the *dnpserver* is declared as failed.

E. Fault Model

Five fault models (see **Error! Reference source not found.**), *bit-flip, adding a constant, byte swap, data overwrite and NOP* (replacing a given instruction with no operation) are employed to study:

(i) the impact of *transient errors* – it is well established that transient errors are mainly caused by cosmic rays, which change the state of a semiconductor by disturbing its electron distribution. Even though many protection mechanisms, such as ECC, parity, and memory scrubbing, are used to cope with transient errors, such errors cannot be fully masked. Previous research on microprocessors [11] showed that logic-level single-bit upset represent 90-99% of device-level transient errors.

(ii) malicious tampering with the data – many attacks are caused by corrupting the control data or non-control data in the memory [10]. Fault models such as *adding a constant, byte swap, data overwrite and NOP* can mimic the consequence of memory corruption attacks.

F. Outcome Categories

The injection outcomes are classified into four categories: *Not Manifested*, *Silent Data Corruption*, *Application Crash*, and *System Hang*, as summarized in TABLE III.

V. EXPERIMENTAL RESULTS

In our experiment, three applications, *monitor_app*, *dnpclient* and *dnpserver*, executing on the substation device RTAC are chosen as the targets for the analysis. A total of 8611 faults are injected into *instructions* and *processor registers* to test the resiliency of the applications. The results are classified by aforementioned outcome categories and analyzed in terms of frequency of occurring, severity, crash causes, and fault types. TABLE IV. summarizes the distribution of different types of injected faults.

TABLE II. FAULT MODELS

Fault Type	Description
Bitflip	Flip single or multiple bits
Add	Increase or decrease the original value by one
Swap	Exchange the high-order bits with the low-order bits
Overwrite	Overwrite the original value with new value
NOP	Replace the original instruction with NOP instruction

TABLE III. OUTCOME CATEGORIES

Outcome Categories	Description
Not Manifested	The corrupted instruction/register is executed or used, but there is no observable impact on application.
Silent Data Corruption	The application does not crash. However, it does not pass the corresponding validation process.
Application Crash	The application crashes due to segmentation fault, illegal instruction, bus error, abort, or exit exception.
System Hang	Application errors propagate to the OS, causing deadlocks of some system resources/services.

TABLE IV. STATISTICS ON INJECTED FAULTS

	Bitflip	***Add***	***Swap***	***NOP***	***Overwrite***	***Total***
monitor_app						
Register	685	633	672	0	608	2598
Instruction	413	441	422	458	396	2130
Total	1098	1074	1094	458	1004	4728
dnpclient						
Register	395	390	390	0	395	1570
Instruction	258	252	275	282	243	1310
Total	653	642	665	282	638	2880
dnpserver						
Register	160	146	123	0	149	578
Instruction	83	104	82	82	74	425
Total	243	250	205	82	223	1003

A. Outcomes Analysis

Key findings from the fault injection experiments are presented in the form of observations and each observation is followed by an in-depth discussion of the causes and consequences of the perceived behavior.

1) Silent Data Corruption (SDC)

Observation 1: *Silent Data Corruption (SDC) can cause an operator in the Control Center to lose control over the equipment in the Substation.*

About 7% of errors (both in registers and instructions) injected in the *dnpserver* and 13% of errors in the *dnpclient* lead to silent data corruption (see Figure 5), i.e., the two applications do not crash but operate incorrectly. SDCs represent the most severe failure category because the system (e.g., SCADA Master or substation devices) operates

using potentially invalid data and hence, any decision based on unreliable data can be incorrect.

In one of the observed cases, ***the dnpserver correctly responds*** to the *Request Link Status*, *Read Data*, and *Breaker Control* commands sent from the Control Center and reports live connection with the Control Center. However, it communicates incorrect data back to the *Control Center*. Worst of all, the *dnpserver* reports that the *Breaker Control* command is accepted, initialized, and queued to be executed while actually the *dnpserver* is not able to successfully pass the command to the *dnpclient*, which in turns should send the command to the *Relay*.

Similarly the ***dnpclient may stay alive*** after an error is injected. However, it stores incorrect or old data in the database (in normal execution, the *dnpclient* aggregates sensor data from *Relays*, and stores the data in the *RTAC* database). In addition, it cannot successfully pass the command from *dnpserver* to the *Relay*. For instance, we observed cases, when an operator in the *Control Center* sends a *Breaker Control* command to the *Relay*, the *dnpserver* receives the command but the *dnpclient* cannot successfully pass it to the *Relay*. At the same time, the *dnpserver* reports that the *Breaker Control* command has completed without confirming whether the command has been really executed.

These scenarios show that detecting SDCs may be hard and its consequences can be catastrophic to the system. For example, consider the case when one of the generators in the power grid fails. The load on other transmission lines may be overloaded. As a result, some load needs to be sheded to prevent damage on the transmission lines. An SDC error can either prevent timely notification of the *Control Center* about the problem, or even if the *Control Center* detects this critical event, it might not be able to control the breaker to avert potential damage of the generating equipment.

Observation 2: *Silent Data Corruptions in the monitoring software (e.g., monitor_app) can make the system unable to recover critical applications from crash failures.*

The job of the *monitor_app* is to monitor and ensure the normal operations of critical applications (*dnpclient* and *dnpserver* in our experiments). When a critical application accidentally crashes, the *monitor_app* restarts it to maintain the availability of the system. The *monitor_app* itself is monitored by the operating system. If *monitor_app* crashes, OS spawns another instance of *monitor_app*. Due to the simple logic of *monitor_app*, the vast majority of injected errors either are benign (not manifested) (52%; see Figure 5) or lead to crashes (45%). In the latter case OS restarts the *monitor_app*. However, about 2% of the injections result in SDC where the *monitor_app* continues to run but is unable to restart the *dnpclient* or *dnpserver* if they crash.

Observation 3: *Propagation of errors can cause deadlocks of system resources*

Propagation of errors in the *dnpclient* can lock some of the system resources. In this scenario, the *dnpclient* does not crash but it cannot pass command to or acquire correct data from the *Relay* (behavior similar to SDC in *Observation 1*). The *monitor_app* does not restart the *dnpclient* because the *dnpclient* is still up and running. What is worse, the console on RTAC hangs and hence, manual termination (e.g., using `kill` command) of the *dnpclient* is not possible. The only solution is to reboot the RTAC, which takes about 20 seconds. The reboot overhead combined with (potentially) long detection latency of the problem may significantly reduce the system availability. Although we could not fully diagnose this failure, the observed behavior is repeatable when injecting the same fault.

2) Crashes and not manifested (benign) errors

Observation 4: *Application crashe dominates the outcome categories.*

Not surprisingly, the most frequent outcome of the fault injection is application crash. Figure 5 shows that about 57% (*dnpclient*), 45% (*monitor_app*), and 54% (*dnpserver*) of outcomes correspond to application crashes. This is because when a fault is injected, it is likely to change the semantic of the instruction causing illegal instruction or change the application control flow causing invalid memory access. Usually, the crash failure is considered less severe than silent data corruption because: (i) the failed application can be restarted (in our settings the *monitor_app* restarts the application) and (ii) error propagation is significantly reduced.

Observation 5: *Percentage of Not Manifested errors in register injections is much higher than that in instruction injections.*

Errors in the code segment are more probable to cause a failure than those in the registers (e.g., in the case of *dnpclient*, not manifested errors contribute to 37% and 17% of the outcome categories for register and instruction injections, respectively; see Figure 6). Corruption of an instruction binary is likely to result in invalid opcode or invalid control flow. In either case, the application is likely to crash. Corruption of register contents must be propagated, i.e., to be used by another instruction, to impact the execution. For example, the following is an instruction which adds register Ra with Rb and saves the result in Rd.

```
add Rd, Ra, Rb
```

Opcode (0-5)	Rd (6-10)	Ra (11-15)	Rb (16-20)	OE (21)	Func (22-30)	Rc (31)

Error in `Opcode` or `Func` changes the semantic of the instruction and an error in `Rd`, `Ra` or `Rb` changes the index of the register. In either case the application is likely to fail. On the other hand, errors occurring in the registers have higher chance to be overwritten. For example, Error in `Rd` would never cause any problem because the register would be overwritten by the instruction result (sum (`Ra + Rb`)).

3) *Crash Cause Analysis*

This section provides a brief discussion on crash causes. **Error! Reference source not found.**and **Error! Reference source not found.** give detailed distribution of crash causes as measured based on the specific exceptions raised by the operating system upon crashes.

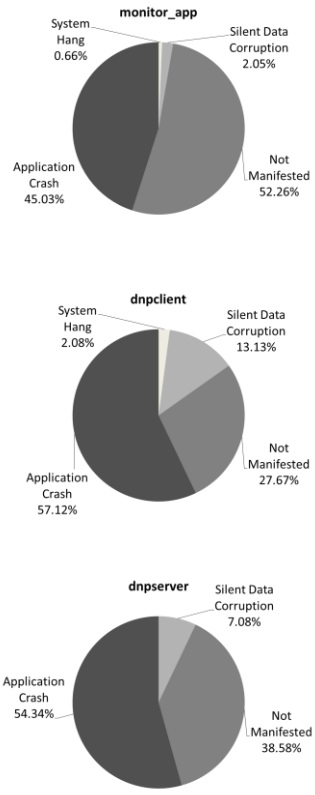

Figure 5: Outcome Category Distribution

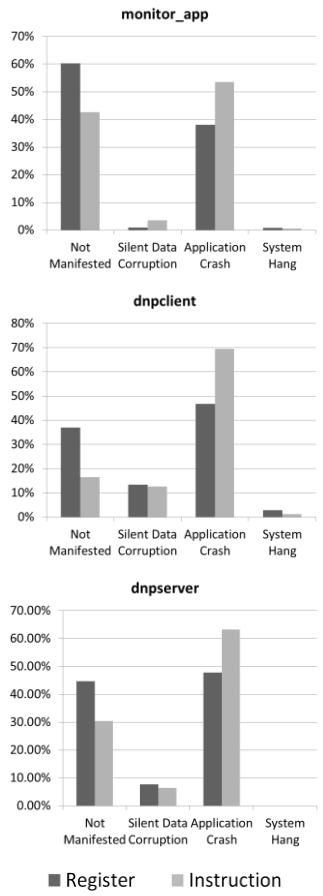

Figure 6: Outcome Categories across Low-level Fault Targets

For all target applications, *segmentation fault* is the most frequent crash cause (65%, 73% and 75% for *dnpclient*, *monitor_app*, and *dnpserver*, respectively). This is not surprising because corrupting memory address stored in the register or the immediate operand in an instruction is likely to result in *segmentation fault* exception.

TABLE V. CRASH CAUSE DISTRIBUTION

	Segmentation Fault	Illegal Instruction	Bus Error	Aborted	App Exit	Others
monitor_app						
Register	974(98.38%)	4(0.4%)	1(0.1%)	4(0.4%)	6(0.61%)	1(0.1%)
Instruction	582(51.1%)	540(47.41%)	0(0%)	1(0.09%)	11(0.97%)	5(0.44%)
Total	1556(73.1%)	544(25.6%)	1(0.05%)	5(0.23%)	17(0.8%)	6(0.28%)
dnpclient						
Register	603(82.15%)	6(0.8%)	15(2.04%)	1(0.13%)	107(14.58%)	2(0.27%)
Instruction	476(52.25%)	359(39.41%)	1(0.11%)	7(0.77%)	64(7.03%)	4(0.44%)
Total	1079(65.59%)	365(22.19%)	16(0.97%)	8(0.49%)	171(10.4%)	6(0.36%)
dnpserver						
Register	263(95.29%)	1(0.36%)	4(1.45%)	5(1.81%)	2(0.72%)	1(0.36%)
Instruction	147(54.65%)	114(42.38%)	0(0%)	3(1.12%)	1(0.37%)	4(1.49%)
Total	410(75.23%)	115(21.10%)	4(0.73%)	8(1.47%)	3(0.55%)	5(0.92%)

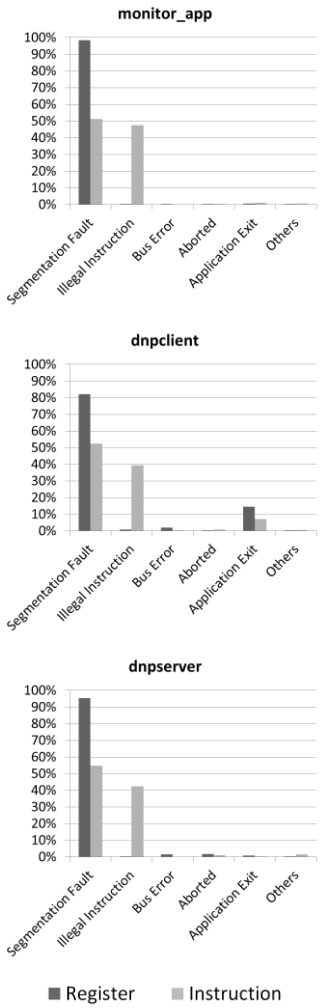

Figure 7: Crash Causes across Low-low-level Fault Targets

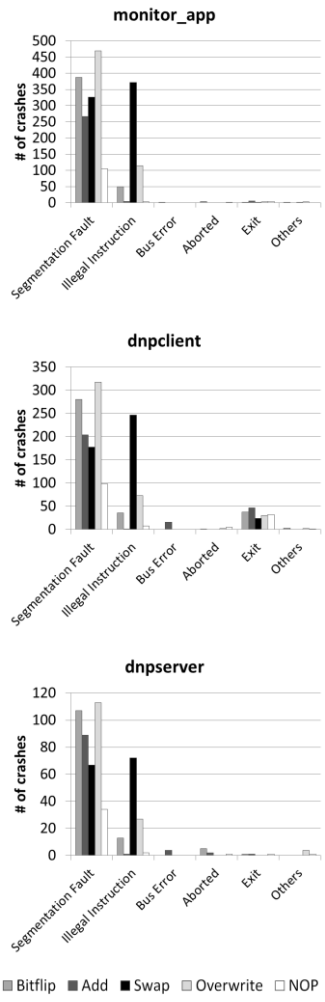

Figure 8: Crash Cause versus Fault Type

Another common reason for application crashe is *illegal instruction*, which accounts for about one fourth of all crashes (see **Error! Reference source not found.**). While the majority of *illegal instructions* are due to corruption of instruction binary, there are cases where register errors cause *illegal instruction*. A closer analysis of error propagation indicates that the latter case happens when the corrupted register is used as a pointer to a code segment. In such case the corrupted register value may point to a memory location that does not store the code but data. The processor attempts to interpret the fetched data as an instruction, which is likely to result in the *illegal instruction* exception.

In addition, we observe that the "application exit" accounts for 10.4% of the crashes for *dnpclient*. As opposed to the

exception raised by the OS, "application exit" means the application itself detects the discrepancy of the data and thus exits with error code. By looking at the faults that cause "application exit", we find that most of the faults that lead to "application exit" are injected into the same function in *dnpclient*, which means this function checks the data integrity to make sure the parameters are within the reasonable boundary before it performs some operations that might affect the system. If the data is unreasonable, the function forces the whole application to exit. We believe that *dnpserver* and *monitor_app* have few "application exit" due to the lack of this kind of checking in their functions. The additional data check implemented by *dnpclient* enables it to exit with error code instead of raising *segmentation fault* exception. If we eliminate these checks from the analysis, all three applications *monitor_app*, *dnpclient* and *dnpserver* have similar statistics on *segmentation faults* (73.9%, 76% and 75.8%, respectively).

4) Crash Cause versus Fault Type

In this section we discuss how the cause of the application crash depends on the fault model. The goal is to analyze the need for different fault models when evaluating the system. **Error! Reference source not found.** reflects the dependency *crash cause vs. fault type*. One can see that all five fault types employed in this study performed well in inducing segmentation faults. However, there are clear differences in the reasons for application failures when different fault models are used.

For example there are cases (see **Error! Reference source not found.** related to *dnpclient*) of *bus error* exceptions caused by using *Add* fault model. The *Add* fault model increases/decreases the register value or the immediate operand of the instruction by one. If the corrupted value is used to address memory, it is likely to cause *bus error* exception due to unaligned memory access. Note that *Bitflip* fault type can also cause *bus error* exception if the low-order bit is corrupted (the change of the memory address must still be in the valid application memory space to prevent *segmentation fault* exception). Another interesting observation is that *Swap* fault type can be very effective in generating *illegal instruction* exceptions.

VI. DISCUSSION

The use of software-implemented fault injection enables us to: (i) conduct inexpensive black-box testing and obtain a good understanding of the interactions between system components (both physical devices and software); (ii) accurately trace error propagation and determine the failure causes; (iii) discover system vulnerabilities, especially the one leading to rare but severe failures; (iv) identify critical variables that, if tampered by attackers, can lead to application crashes or Silent Data Corruption (Protecting these critical variables from tampering can reduce the possibility of successful attacks); (v) provide useful feedback for the developers and vendors to improve the security and error resiliency of devices.

Detecting Silent Data Corruption is hard because the erroneous application may not crash but work incorrectly. In one of the observed SDC scenarios, the operator in the *Control Center* requests sensor data from the *Substation* and the *dnpserver* responds to the request. However, it sends incorrect data back to the operator. The operator has no way to directly determine whether the data is correct unless there is an alternative way of verifying the correctness of the data received from the *Substation*. Worst of all, the operator cannot easily determine where the errors occurred because the communication path between *Control Center* and the *Substations* involves many different devices and networks. The operator needs to check all devices and networks along the path to determine the root cause. Potential solution would be to gather redundant data through another communication channel or from a neighboring *Substation* to detect the inconsistency of data.

VII. CONCLUSIONS

This paper evaluates the robustness of power grid substation devices in the face of transient errors and malicious attacks. As a case study, faults/errors are injected to the instructions and processor registers of a Real Time Automation Control device in a *Substation*, which plays an important role as a communication bridge between the *Control Center* and the *Substation* in the power grid infrastructure. The injected disturbances mimic the impact of transient errors and malicious attacks on applications executing on the devices. Our result indicates that an error in control software may cause the operator in the *Control Center* to lose control over the breaker monitored by Relay in the *Substation*. Consequently, precautions must be taken to ensure robust and continuous operation of the devices.

VIII. ACKNOWLEDGEMENT

This material is based upon work supported by the Department of Energy under Award Number DE-OE0000097.

REFERENCES

[1] U. S. Department of Energy. "Top 10 vulnerabilities of control systems and their mitigations – 2007." Online, Decemeber 2006.

[2] North American Electric Reliability Corporation (NERC). "Results of the 2007 survey of reliability issues." Online, October 2007.

[3] W. Gu, Z. Kalbarczyk, and R.K. Iyer, "Error sensitivity of the Linux kernel executing on PowerPC G4 and Pentium 4 processors," in Proc. of Conference on Dependable Systems and Networks, 2004.

[4] C. Pham, D. Chen, Z. Kalbarczyk, and R.K. Iyer, "CloudVal: A framework for validation of virtualization environment in cloud infrastructure," DSN, 2011.

[5] D.T. Stott, G. Ries, M. Hsueh, and R.K. Iyer, "Dependability analysis of a high-speed network using software-implemented fault injection and simulated fault injection," in IEEE Trans. on Computers, 1998.

[6] Idaho National Laboratory. "Vulnerability anaylsis of energy delivery control systems", Online, September 2011.

[7] L. Mili and Q. Qiu, "Risk assessment of catastrophic failures in electric power systems," Int. J. Critical Infrastructure, vol. 1, 2004.

[8] Z. Anwar, R. Shankesi, and R.H. Campbell, "Automatic security assessment of critical cyber-infrastructures," DSN, 2008.

[9] A. Faza et al., “Integrated cyber-physical fault injection for reliability analysis of the smart grid,” SAFECOMP, 2010.

[10] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. Iyer, “Non-Control-Data Attacks Are Realistic Threats,” USENIX, 2005

[11] M. Rimen, J. Ohlsson, and J. Torin, “On Microprocessor Error Behavior Modeling,” Proc. of FTCS-24, 1999.

[12] T. Rigole, K. Vanthournout, and G. Deconinck, “Resilience of Distributed Microgrid Control Systems to ICT Faults,” CIRED, 2007.

[13] D. T. Stott et al., “NFTAPE: a framework for assessing dependability in distributed systems with lightweight fault injectors,” IPDS, 2000.