

Error Propagation Analysis Studies in a Nuclear Research Code

Jeffrey Voas, Frank Charron
Reliable Software Technologies
Suite 250, 21515 Ridgetop Circle
Sterling, VA 20166
(703) 404-9293
{jmvoas,fhchar}@RSTcorp.com

Leo Beltracchi
US Nuclear Regulatory Commission
MS: T10-E33, 11555 Rockville Pike
Rockville, MD 20852
301-415-6658
lxb@nrc.gov

Abstract—Software fault injection has recently started showing great promise as a means for measuring the safety of software programs. This paper shows the results from applying one method for implementing software fault injection, data state error corruption, to the Departure from Nucleate Boiling Ratio (DNBR) code, a research code from the Halden Reactor Project, located in Halden Norway.

THE OPINIONS AND VIEWPOINTS PRESENTED ARE THE AUTHOR’S PERSONAL ONES AND THEY DO NOT NECESSARILY REFLECT THE CRITERIA, REQUIREMENTS, AND GUIDELINES OF THE U.S. NUCLEAR REGULATORY COMMISSION.

TABLE OF CONTENTS

1. INTRODUCTION
2. FAULT INJECTION FUNDAMENTALS
3. EXPERIMENT USING NUCLEAR CODE
4. CONCLUSIONS

1. INTRODUCTION

If we knew what tomorrow held, then we could know precisely how good a specific piece of software is today. We would know when in the future it will fail, how often it will fail, and under what conditions it will fail. Armed with that information, it would be trivial to confidently assess the “goodness” of the code.

To us, the future is a *black box*. We do not know what faults will be triggered nor what anomalous input data will be sampled, and we do not know how severe the failures will be. All that can be done from a development standpoint is to fix *known*, existing faults and boost the software’s fault tolerance to external anomalies; but since we cannot get out all of the faults nor protect against all expected anomalies, traditional methods of protection are unsatisfying.

Fortunately, we are not limited to traditional methods of measuring software quality. We can estimate the “behavioral goodness” of the code under predicted *anomalous* circum-

stances that could arise tomorrow. This provides a means for predicting the quality of the code under simulations of undesirable circumstances. The inability to accurately predict future behavior has long been a problem for software quality metrics. The problem has been an inability to predict behavior when faced with *unexpected* situations. However, fault injection is a completely different breed of metric, which, because of its more complex nature, can partially address this problem.

Software fault injection includes a family of techniques that instrument the original code with mechanisms (more code) that either modify the existing syntax or force the state of the code to be modified when the software executes. Either way, the code or its behavior is changed. It is this process of modifying events that makes it ideal for predicting what might happen if future events get disturbed. In this paper, we will discuss results that were achieved by modifying internal data states in a nuclear program written at the Halden Reactor Project (Halden Norway).

2. FAULT INJECTION FUNDAMENTALS

Development processes, even formal ones, are not solutions for *measuring* software quality; they only seek to build software with quality behavior. Fault injection, a process that doesn’t tell how good the code is *per se*, but instead provides worst case predictions for how badly the code might *behave* in the future (as opposed to how bad it might *be*) is a nice compliment to any set of development processes. Recognize that correct code can even have “bad days” and not work as desired due to external influences on it. For example, your workstation might be in perfect working order, but if the network server is not functioning, you may find the workstation so unusable that it might as well be broken too. The worst case predictions from fault injection are not a *direct* predictions of the future, but instead *indirect* foreshadowings of how bad the future could get.¹

¹Software testing might be able to give a direct foreshadowing of how

If software does not experience any problems (such as exercising internal faults or receiving corrupt input data) during execution, then the software cannot behave badly; only when software encounters problems that corrupt its program state can things then go awry. An *anomalous operational scenario* (or anomaly) is an event that creates a data state during software execution that alters the output of the software such that the output is undesirable. The number of different anomalies that software can experience during its lifetime is almost always infinite and unknown.

The anomalies that fault injection is capable of simulating are those that can arise from code defects (caused by programming errors or design defects), human factor errors, or other external failures (from hardware upon which the software depends for inputs or from other software). The combinations of anomalies from instances of these different problem sources is intractable. When fault injection is employed as a means of measuring *testability*, as in Voas's PIE model [8], code defects are simulated. When fault injection is employed for measuring *safety* or *failure tolerance*, human factor errors and other external forms of failure are simulated. Programmer faults need not be simulated by fault injection because the actual programmer faults are already in the code. Adding new hypothesized programmer faults could bias the results. But even ignoring programmer faults, the union of all human factor errors and other external anomalies is still a very large set of anomalies. We must limit this set to the most relevant anomalies because there is no way that we could simulate all of them.

Software standards provide some help here. Software engineering standards sometimes enumerate certain classes of problems that must not manifest during software execution. The argument *for* doing so is that you can't protect against problems until you know what the problems might be. Here, problems are usually defined in one of two ways: (1) a class of failure that must not occur, or (2) a fault class that can occur but the fault class must be shown to only cause acceptable outputs.

But simply having software standards does not ensure that all of the enumerated problems will be protected against. You still need a way to measure how well the developer implemented the appropriate protection. Demonstrating that software outputs are what we want (or at least can tolerate) requires showing that even if bad events happen during computation, undesirable outputs will not.

Fortunately, this demonstration can be partially accomplished via fault injection methods, since we can observe how *well-behaved* software is in even the most disheartening of circumstances. Voas *et. al.* [10] provide a discussion for deciding

how to select the anomalies that should be injected in order to predict worst case outcomes. Once these outcomes are observed, we can build protection schemes into the software in order that those behaviors cannot be exhibited by the software in the future.

3. EXPERIMENT USING NUCLEAR CODE

Previously, several real-world case studies have been published that showed the benefits of fault injection analysis, one for a medical device, and the other for a train control system [1]. In [9], the Therac-25 (See [7]) and Ariane-5 (See [2]) incidents were reviewed, and hypothetical analyses were provided for how fault injection could have warned of those problems before the software was deployed. (Interestingly, those two incidents could have been avoided had any of a wide variety of common software engineering techniques been employed, not only fault injection.)

Generally speaking, software fault injection should *not* be viewed as a debugging technology. Because fault injection has no access to a test oracle for the software and only requires access to information concerning which output states are hazardous, the only fault-detection that fault injection can do is to warn if the safety monitoring code inside the software is working properly or not (assuming that the hazards you define for the fault injection tool are the correct ones).

Thus instead of viewing fault injection as a debugging technology, fault injection should be considered as an analysis technique that provides insight into the frequency with which hazardous software outputs occur given that certain corrupted internal states are created after the software is deployed. Even if fault injection observes hazardous output after it injects the anomalies, fault injection does not go the next step and assert that the software *will* definitely cause hazardous output in the future, because fault injection cannot guarantee that the software will get into the corrupted states that fault injection employed in the future.

Software fault injection simply asks "what if?" this anomaly were to occur and then answers the question. As we said, there are certain cases where faulty safety monitoring code can be detected using fault injection, and fault injection is capable of pinpointing where those errors exist. In this section, we will explain a nuclear application, the Departure from Nucleate Boiling Ratio (**dnbr**) program, where this interesting "debugging" phenomenon occurred and the safety monitoring code was found to be faulty.

The **dnbr** program was obtained from the Halden Reactor Project. The Halden Reactor Project is an international research organization with member organizations from 19 countries around the world, including the US Nuclear Regulatory Commission. A objective of the research activities is safe and efficient operation of nuclear power plants. Al-

bad the future could get, if operational profiles and anomaly frequencies are available, but this is debatable.

though the **dnbr** program was made as a realistic program, its sole purpose was to be a testbed in order to conduct research on software verification and validation. It is therefore not currently running or intended to be implemented on any nuclear power plant.

Although it is a test program, the **dnbr** program was made to reflect a realistic implementation in a nuclear power plant. This realistic scenario consists of the following parts:

1. A set of process data are read from in-core instruments and converted to a digital form. This data consists of absolute pressure, pressure difference between top and bottom, temperature and neutron flux from four set of triplet redundant flux detectors. Table 1 shows the data ranges of the input data.
2. These data are at regular time interval read by a Data Validation program. This program makes a variety of checks on the input data to identify any faults in the in-core instruments. Information on faults is sent to an Error Handler program. However, due to redundancy, the **dnbr** program can tolerate faults in the flux detectors.
3. The (corrected) process data are transferred to the Dnbr Main program, which computes the departure from nucleate boiling ratio and checks this against a safety limit. If this limit is exceeded, a trip signal will be given which activates a reactor shut-down.
4. As the data validation uses process data from the previous execution cycles, an initiation program computes and stores initial values for all process variables.

In the testbed, the process data are simulated within the input ranges shown in Table 1, using a generator for pseudo-random numbers. The data generation, data validation and **dnbr** checking are executed in a loop, and the output is stored. The output consists of the trip signal and the error messages, as well as of the process data transferred from Data Validation to Dnbr Main, and the value of the **dnbr** ratio. The latter was intended for detection of masked faults in the program. The process data were also stored internally to be used in data validation in the subsequent cycle.

Results

Fault injection has a variety of applications, however from previous experiences, safety assessment has turned out to be an application where fault injection seems to be best suited. To observe whether code can output unsafe states after anomalies are injected, unsafe output states must first be defined. For **dnbr**, two classes of unsafe (hazardous) output states were identified:

1. Missing identification of instrument failures, due to faults in the Data Validation program.

2. No trip signal when required, due to faulty computation of the burnout in the **dnbr** main program.

We now discuss these hazardous states and the data we collected telling how often we were able to observe unsafe output.

*Hazard One: Fault Identification of Instrument Failures—*The software tests for the first type of software hazard via *drastic time check* code (which is simply the data validation safety monitor that is implemented in the software). There are tests within the **dnbr** program to test the magnitude of changes in each input variable from previously measured values. The purpose of this monitor is to ignore input variable data if it exceeds a physical limit of change from the last time step. Tests for this are performed in the **drastic.time.check** routine. When the program's state is tested, if a value fails the check, a global variable **pass_drastic.time.check** is set to indicate a hazardous situation has arisen.

The basic control flow of the drastic time check implementation is detailed in pseudo-code in Figure 2. During fault injection, a specific hazard was instrumented at each line where a star ('*') appears in the pseudo-code.

Note that an '*' does not appear in this pseudo-code for the test of the **PRESSURE** value. In our first attempt at inserting the appropriate hazard-detection monitor for **PRESSURE** (in **_drastic.time.check_**), the monitor starting warning of hazardous behavior on every test case *before* anomaly injection occurred. This indicated that a fault already existed. So to analyze the software with respect to the remaining hazards, we removed the hazard monitor on **PRESSURE** value.

Hazard-detection monitors were then placed on the remaining three checks and fault injection was performed. When the results were analyzed, we discovered that the code portions responsible for the checks for: (1) *pressure difference*, (2) *temperature*, and (3) *neutron flux were not exercised*. But the check for *pressure* was exercised. This unusual circumstance immediately raised suspicions.

After further analysis, a programmer fault was discovered in the conditions that determine which type of drastic time check to apply. In the pseudo-code above, you can see that there are four different types of drastic time checks: (1) pressure, (2) pressure difference, (3) temperature, and (4) neutron flux. The pressure difference, temperature, and neutron flux conditions that test for hazardous data were executed based on comparisons using **_value_**; whereas the pressure test is executed based on a comparison using **_type_**. This is an *error* for the pressure drastic time check. The programmer introduced a fault by comparing **_value_** instead of **_type_** in order to determine which drastic time check to use. Various consequences of this error will now be detailed.

It is very unlikely that the value passed in **_value_** will be

Variable	Type	Input Range
input_space.pressure	int	[14400, 17600)
input_space.pressure_difference	int	[13300, 20000)
input_space.temperature	int	[16400, 17200)
input_space.channel[i][j], $0 \leq i < 4, 0 \leq j < 3$	int	[7730, 10450)

Figure 1: Input data ranges for **dnbr**.

equal to the correct value of its associated type parameter (i.e. **PRESSURE_DIFF**). In fact, it is more likely that `_value_` will contain a value that does not match any of the four types, in which case `_drastic_time_check_` fails to apply any checks, i.e., the software mechanisms provided in the code to check for safety will behave as if they were not in the code. Since the input data is rarely checked for pressure difference, temperature, and neutron flux measurements, invalid data will likely go undetected. This defeats the purpose of having those checks in the code, and this is the case where checks that are needed are not being performed.

The pressure difference check tests the input variable `_value_` for excesses to the physical limits defined in pressure difference. The physical limits specific to the other types are tested in their respective drastic time checks. A value that is acceptable according to the drastic time check of its associated type may be unacceptable according to the drastic time check of a different type. As an example of this, suppose that `_drastic_time_check_` was called to perform the (**TEMPERATURE**) drastic time check. The error in the expression

```
(_value_ == PRESSURE_DIFF)
```

may cause the pressure difference (**PRESSURE_DIFF**) drastic time check to be applied instead. The incorrect type check may result in the detection of some truly invalid values; however, it is also possible that the check will result in falsely firing a failure signal for a valid (`_value_`, `_type_`) pairing. Therefore, it is possible that `_value_` can enter `_drastic_time_check_` in such a way that it is valid according to `_type_`, but is determined to be hazardous because of this fault. So this represents the case where checks that are not needed are performed.

Hazard Two: No Trip Signal When Needed—The software tests for the second hazard via a *burnout* assertion (which is simply a safety monitor that is implemented via software and embedded in the nuclear application). Burnout occurs during a transition from a nucleate boiling state, a *safe* condition of operation, to a film boiling state, an *unsafe* condition of operation. Failure to tell us when this transition has occurred presents a serious problem.

In the original code supplied by the Halden Reactor Project, a safety monitor already existed for this condition:

```
if (Q[j] > (0.7 * QBO))
    gbl_data.trip_signal = TRUE;
```

If this condition is true, the system is considered as being in a hazardous state. (The multiplier 0.7 is used in the **dnbr** code to provide a safety margin to the actual cross-over threshold.) Using the above hazard with the 0.7 multiplier, we did not observe hazardous output after fault injection inserted anomalies. With 0.7 as the multiplier, the code demonstrated high fault tolerance.

As an experiment, we then decided to try to find a new multiplier (hence a new hazard) such that the new hazard would occur. To do this, we started decreasing the multiplier to obtain the sensitivity of the **dnbr** program to multipliers that were smaller than 0.7. The multipliers tried are shown in Table 2, and as we continued to decrease the multiplier, we were finally able to force a hazard to trigger.

What was actually at work here is this: for a fixed value in $Q[j]$, and by gradually decreasing the multiplier, we were increasing the likelihood of a hazard warning occurring (from our embedded hazard observer). Interestingly, the “modified” hazard tests that we injected that employed a multiplier of 0.27 or greater to check for burnout were never triggered by any injected corruption. Given the test cases and injected anomalies, this suggests that if the multiplier is greater than or equal to 0.27, the program is very tolerant to anomalies.

For multiplier values between 0.27 and 0.21, we observed a small handful of hazard warnings being triggered, but not until the multiplier was 0.21 or less did hazard warnings occur frequently. With smaller multipliers, the calculations made in **dnbr_MAIN** are much more sensitive to our injected anomalies. This indicates that if the actual multiplier in the safety monitor of **dnbr** were to be modified to a value around 0.21, then failure tolerance assertions and data checks should be made at the intermediate calculations in the **dnbr_MAIN** code, because at the lower multiplier values, hazard warnings will be frequently going off. By adding failure tolerance assertions to trap anomalous internal states, those states will not be allowed to propagate through the succeeding calculations to the burnout condition (hazard). This will help eliminate the number of false positives and false negatives reported by the **dnbr** program’s actual burnout check.

Recognize that nothing is known to be wrong with a multiplier value of 0.7 in the burnout hazard. But also, it is not known if 0.7 is the “best” value that could be used in the burnout assertion. Since we could not trigger the burnout assertion in its original form, we simply “played games” until

```

FUNCTION: _drastic_time_check_
INPUTS: int _value_, int _type_
{
  IF (_type_ == PRESSURE) {
    IF (_value_ passes pressure drastic time check) {
      SET a flag
    } ELSE {
      PRINT error message
      DO error handling
    }
  } ELSE IF (_value_ == PRESSURE_DIFF) {
    IF (_value_ passes pressure difference drastic time check) {
      SET a flag
    } ELSE {
      PRINT error message*
      DO error handling
    }
  } ELSE IF (_value_ == TEMPERATURE) {
    IF (_value_ passes temperature drastic time check) {
      SET a flag
    } ELSE {
      PRINT error message*
      DO error handling
    }
  } ELSE IF (_value_ == NEUTRON_FLUX) {
    IF (_value_ passes temperature drastic time check) {
      SET a flag
    } ELSE {
      PRINT error message*
      DO error handling
    }
  }
}
}
}

```

Figure 2: Pseudo-code for the drastic time check.

we were able to find assertions where the injected anomalies started triggering the assertions. This was an academic exercise that served a useful task however; it tested the sensitivities of different hazards. If it were the case that it is unsafe to have unnecessary warnings going off, an analysis like this can help find the multiplier threshold where unnecessary warnings are more likely to start occurring. This will allow for higher or lower safety margins, while ensuring that unwarranted warnings are not produced.

Future Experiments Involving Sensor Sensitivity—One other useful application of fault injection is to measure software’s sensitivity to incoming sensor data. In the **dnbr.c** code there is a statement where this could be performed:

```
tsat = 1804 + (DNBR_data.pressure/30.0);
```

tsat is the saturation temperature for water, and tsat is a function of input data contained in the variable, DNBR_data.pressure. Simply stated, tsat is the temperature at which water boils. If (1) the water entering the reactor is near the boiling point, (2) the flow is low, and (3) heat

by the reactor high, then the conditions are ripe for burnout and film boiling.

By simulating the pressure sensor making inaccurate measurements due to calibration errors, fault injection could be used to quantify the degree of sensor error that could lead to unnecessary or incorrect burnout warnings occurring if **tsat** were incorrectly calculated. We have not yet performed this experiment.

There is yet another experiment that we could have performed using the above assignment statement. This experiment would test the software’s behavior to imperfections in the linearization. The experiment would simulate imperfections in the calculation by moderately corrupting **tsat** (*i.e.*, making very small modifications to its value) after the assignment statement in its original form is executed. This tells us how severe corruptions of **tsat** can be while still not triggering a hazard.

Replacement Hazards
$(Q[j] > (0.29*QBO))$
$(Q[j] > (0.28*QBO))$
$(Q[j] > (0.27*QBO))$
$(Q[j] > (0.26*QBO))$
$(Q[j] > (0.25*QBO))$
$(Q[j] > (0.24*QBO))$
$(Q[j] > (0.23*QBO))$
$(Q[j] > (0.22*QBO))$
$(Q[j] > (0.21*QBO))$
$(Q[j] > (0.20*QBO))$
$(Q[j] > (0.19*QBO))$
$(Q[j] > (0.18*QBO))$
$(Q[j] > (0.17*QBO))$
$(Q[j] > (0.16*QBO))$
$(Q[j] > (0.15*QBO))$

Figure 3: Our “modified” hazards to test for a better multiplier.

4. CONCLUSIONS

Software fault injection is an emerging technology that can be used to observe how software systems behave under experimentally-controlled, anomalous circumstances. Results from software fault injection act as a crystal ball, predicting how badly software will behave should things go awry (both internally and externally) during execution. Such predictions provide data as to how robust a piece of code is, where in the code failure tolerance is deficient, and most importantly, what level of risk is incurred by relying on a particular software system. In short, fault injection is an efficient, scientific means for predicting how events might unfold in the future.

Portions of the **dnbr** program were analyzed using fault injection. Fault injection was able to reveal a true logical fault that was coded incorrectly at three different conditions in the safety monitoring code. The tool also provided results that describe the sensitivity of the software’s tolerance to the tightening of the burnout hazard condition.

The underlying ideas behind fault injection are not new, and there exists much literature describing how to employ them for hardware system validation, software testing, and hardware design validation [3,4,5,6,8]. Unfortunately, the migration of those ideas into practical methods for software validation has not occurred, mainly due to concern about the plausibility of the anomalies injected. For example, in an integrated circuit, the failure classes are obvious: stuck-at-one, stuck-at-zero, etc. But for a software system, the internal data states are not as simple as just a ‘0’ or ‘1’, and hence the number of

different anomalies that could be injected is intractable.

As software *liability* becomes a more pressing concern, having one more chance to make improvements can provide added “peace of mind” [10]. Most prudent development organizations are diligent to pursue traditional methods of assurance, and that provides some degree of “peace of mind.” For example, testing the software heavily under *expected* circumstances is one way to perform system reliability testing. In contrast, fault injection provides a way to test using *unexpected* circumstances. This adds confidence in the code that reliability testing cannot provide. In short, this combination of reliability testing and fault injection are not redundant effort, rather complimentary effort. And finally, there is no way to avoid software liability; there are, however, ways to reduce uncertainty, and software fault injection is one such way.

ACKNOWLEDGEMENTS

The authors thank Gustav Dahll, Björn Axel Gran, and Harald Thunem of the Halden Norway Nuclear Reactor Project (Halden, Norway) for providing access to the DNBR code. The authors also thank Gustav Dahll for his comments on an earlier draft of this manuscript. This work has been partially supported by DARPA Contract F30602-95-C-0282 and NIST Advanced Technology Program Cooperative Agreement No. 70NANB5H1160.

REFERENCES

[1] J. Voas, F. Charron, G. McGraw, K. Miller, and M. Friedman. Predicting How Badly ‘Good’ Software can Behave.

IEEE Software, 14(4):73–83, July 1997.

[2] Prof. J. L. Lions. Ariane 5 flight 501 failure: Report of the inquiry board. Paris, July 19, 1996, available at http://www.cnes.fr/actualities/news/rapport_501.html.

[3] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J.-C Fabre, J.-C Laprie, E. Martins, and D. Powell. Fault Injection for Dependability Validation: A Methodology and Some Applications. *IEEE Trans. on Software Engineering*, 16(2):166–182, February 1990.

[4] J. A. Clark and D. K. Pradhan. Fault Injection: A Method for Validating Computer-System Dependability. *IEEE Computer*, pages 47–56, June 1995.

[5] J. A. Solheim and J. H. Rowland. An Empirical Study of Testing and Integration Strategies Using Artificial Software Systems. *IEEE Trans. on Software Engineering*, 19(10):941–949, October 1993.

[6] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34–41, April 1978.

[7] N.G. Leveson and C.L. Turner. An Investigation of the Therac-25 Accidents. *IEEE Computer*, 26(7):18–41, July 1993.

[8] J. Voas. *PIE*: A Dynamic Failure-Based Technique. *IEEE Trans. on Software Eng.*, 18(8):717–727, August 1992.

[9] J. Voas. Software Fault Injection: Growing Safer Systems. In *Proc. of IEEE Aerospace'97*, Snowmass, CO, February 1997.

[10] J. Voas, G. McGraw, L. Kassab and L. Voas. A Crystal Ball for Software Liability. *IEEE Computer*, 30(6):29–36, June 1997.

Jeffrey Voas is a Co-founder and Chief Scientist of Reliable Software Technologies and is currently the principal investigator on research initiatives for DARPA and the National Institute of Standards and Technology. Voas has also recently served as a Principle Investigator on efforts for NASA, National Science Foundation, and the USAF. He has published over 85 refereed journal and conference papers. Voas has coauthored a text entitled *Software Assessment: Reliability, Safety, Testability* (John Wiley & Sons, 1995). Voas is currently co-authoring a second text entitled "Software fault-injection: inoculating programs against errors", due to be published by Wiley in 1997. Voas will be the special editor of an *IEEE Computer* theme issue on "Commercial off-the-shelf software" in 1998. Voas was the General Chair for COM-PASS'97, and serves on the Editorial Board for the *Software Quality Professional Journal*. Voas will serve as the Program Chair of ISSRE'99. Voas is currently writing a book chapter

on software liability for the "Advances in Computer" book series. Voas's current research interests include: information security metrics, software dependability metrics, software liability and certification, software safety and testing, and information warfare tactics.

Voas is currently a consultant to Hughes on the FAA's new Wide Area Augmentation System (WAAS) Air Traffic Control project. In 1994, the *Journal of Systems and Software* ranked Voas 6th among the 15 top scholars in Systems and Software Engineering. Voas is a member of IEEE and received a Ph.D. in computer science from the College of William & Mary in 1990.

Frank Charron is a Research Associate at Reliable Software Technologies. He serves as Project Leader of the RST WhiteBox SafetyNet commercial product. SafetyNet is a fault-injection software tool that can be used to assess the fault-tolerance of applications. Mr. Charron has worked with several clients in applying SafetyNet and fault injection to analyze safety-critical systems. Mr. Charron also is currently leading up the development of RST's new research tools in the security domain, including the Fault Injection Security Tool (FIST). FIST is a fault-injection tool that simulates malicious data states during execution of an application and determines whether they ultimately lead to a violation of the system's security policy. The FIST tool has been used to measure the vulnerability of several networked based applications, including HTTP and FTP server programs.

Mr. Charron has an M.S. in Operations Research from The George Washington University and a B.A. in Mathematics from the University of Virginia.