Detection, Analysis, and Prediction of the Effects of Electrostatic Discharge on USB Peripherals

Nathan Jarus, *Member, IEEE* Antonio Sabatini, *Member, IEEE* Pratik Maheshwari, *Member, IEEE* and Sahra Sedigh Sarvestani, *Member, IEEE*
Department of Electrical and Computer Engineering
Missouri University of Science and Technology, Rolla, MO 65409, USA
Email: {nmjxv3,ajs5gd,prm8c7,sedighs}@mst.edu

***Abstract*—Understanding the effects of electrostatic discharge (ESD) on embedded systems and their peripherals is useful for both hardware and software design. Manufacturers must design hardware that can withstand ESD during normal operation. They may also wish to know which components on a defective board have been damaged by ESD in order to repair the system. Software developers may want to ensure that the software controlling these peripherals can quickly recover from an ESD event without crashing.**

In order to instrument commercially produced hardware, we develop a software-based ESD monitoring methodology requiring no special hardware on the embedded system. This methodology can be applied to any embedded system peripheral and allows the system to operate normally while the monitoring software is running. We apply our methodology to a USB host controller on an embedded system and explain the significance of the registers we monitor. Furthermore, we develop a classifier capable of determining whether or not the system is experiencing ESD based on its current state.

***Keywords*—Software instrumentation, Electrostatic Discharge, Failure Analysis, Universal Serial Bus, Computerized Instrumentation, Embedded Software, Software debugging, Software design, Classification algorithms, Monitoring, Registers**

I. Introduction

As embedded systems become smaller and smaller, they become more vulnerable to physical events and thus more difficult to make reliable. Electrostatic Discharge (ESD) is a major cause of this unreliability, since smaller components require a smaller electrical charge to experience an ESD event. The effects of these events on the software running on the embedded system are not well understood yet. In order to understand these effects, we must observe how the hardware effects of ESD appear to the software controlling that hardware.

Some ESD events do not cause permanent damage to the hardware but instead cause software glitches that appear random and unexpected to the system user. Hardware-level analysis methods struggle to monitor these types of events; it is difficult to associate a user-observed event with a specific physical event. Additionally, component miniaturization is increasing the difficulty of monitoring all traces on a board for ESD. Even if all the traces could be monitored, analyzing how the ESD coupled to the hardware and what effects it had on various components is challenging. Finally, while invasive hardware testing might be feasible on a development board, testing on consumer hardware not designed for such testing is much more difficult.

In order to avoid these issues with hardware ESD analysis, we propose a low-level software analysis method. Software instrumentation allows hardware that cannot be physically probed to be observed during ESD events. However, many software analysis techniques focus on high-level faults instead of hardware failures. Other software approaches study low-level failures, but run on the bare metal, preventing the embedded system from being used as it was intended. Our approach uses modified hardware drivers to allow a fully-functioning system to be monitored for ESD events. In our work, we consider small-scale ESD events that do not cause permanent system harm.

With software instrumentation, we are able to observe the changes in system state caused by ESD. From this data, along with data concerning normal system operation, we can construct software capable of predicting when a system is experiencing ESD. These results have several applications in hardware and software design, as well as failure analysis. Throughout this paper, we will discuss these observations and analyses within the context of a USB host controller on an embedded system.

Our research contributions are:

- A method for instrumenting device drivers to monitor peripheral operation.
- A method for combining and relating the observed states of peripheral operation.
- A classifier capable of estimating whether or not the system is currently experiencing ESD.

The rest of the paper is as follows:

- An overview of related work.
- A description of the software instrumentation approach.
- An overview of the hardware testing and measurement process.
- A discussion of the experiments' results.
- A description of the log classification and state prediction algorithm.

II. Related Work

Hardware ESD fault injection with direct injection and field injection probes is performed in [1], [2], [3], and [4].

These studies characterize integrated circuit (IC) immunity to ESD. The sensitivity threshold for each IC was determined by injecting ESD at increasing voltages and observing when errors occurred. In these studies, only user-visible errors, such as screen glitches or hardware resets, were studied.

[5] and [6] extend this fault injection process by mapping the ESD sensitivity of the board. The injection probes are attached to a 3-D scanner that can sweep them across the board. ESD is injected at each point on a grid and the device's susceptibility to ESD at each point is noted. The resulting map can be used to identify traces and components that are at risk for ESD damage.

[7] takes a design-time approach to this problem. They develop a tool capable of analyzing circuit designs for ESD susceptibility that guides the optimal placement of sensitive components and traces. A similar method that encompasses simulations of ICs, IC packaging, printed circuit board (PCB) traces, and passive components is presented in [8]. The authors verify their model with a test PCB that allows them to inject ESD on specific traces and components. Unfortunately, analytic approaches such as these become prohibitively complex on modern circuit layouts.

While not directly related to ESD events, software-based and combined hardware and software system monitoring has been studied extensively. [9] outlines research related to monitoring for runtime verification. The system state is monitored by some combination of hardware and software; this information is then used to verify that the system is operating within specification. A software-specific study of fault monitoring is presented in [10]. The authors present a taxonomy of runtime monitoring approaches and discuss various system requirements for different monitoring techniques. [11] provides a tutorial in designing and implementing on-line monitoring systems.

[12] develops a mixed hardware and software approach for logging nondeterministic behavior in embedded systems. They modify the compiler for an embedded system to emit code that logs messages to an attached system that stores those messages. This combined approach greatly reduces processing overhead on the monitored system, making it applicable to very low power embedded hardware. [13] creates a tool that takes a software specification for an embedded system and produces both an executable that perform that specification and a configuration for a hardware monitor. The hardware monitor interfaces with the embedded CPU and its communication buses and verifies the operation of the system. [14] uses the system management mode feature of Intel x86 CPUs to allow an external system to monitor the behavior of applications. The external system reads the memory ranges occupied by specific software, such as a web browser, and looks for behavior that may indicate malware. [15] extends this concept to an entire cluster of machines. Their hardware monitors a shared communication bus and records events and can be used for debugging, monitoring, and fault recovery.

The SIPE project [16] laid the foundations for software monitoring of computer systems. It instrumented an event-driven multitasking system, recording the performance of the scheduler, the CPU, and various peripherals. [10] develops a software-based monitoring system for ATMs. They instrument each piece of hardware with kernel drivers that measure hardware state and performance. The data from these sensors are filtered and a runtime checker uses the filtered data to determine if the system is operating correctly. If not, recovery actions can be taken to restore system availability. [17] augments log-based rollback-recovery to support software interrupts in Log-based recovery is based on the software operating in a completely deterministic fashion based on which messages it receives. These messages are logged, so if the system fails, it can resume from the last message logged without any state inconsistencies. To expand their model to asynchronous operations and multithreaded software, the authors instrument a threading library to log additional data, recording the nondeterminism caused by thread-based operations. They analyze the performance impact of this logging and verify that it incurs little overhead. [18] presents seven patterns for improving embedded system reliability with watchdog timers. A watchdog timer is a hardware timer that restarts the system if the timer is not reset within a specific time. Depending on what recovery behavior is desired and what sort of programs are run on the system, different approaches to 'feeding' the watchdog are taken.

Of particular interest is the software black-box failure analysis method. [19] develops a recorder that stores system state transitions as the system operates and a decoder that evaluates the recorded transitions and determines which software module the system failed in. Their software was capable of detecting some failures and capturing causes for some of those failures. Overall, they discovered that the granularity level at which they recorded system states was too coarse to detect certain failures. However, increasing the number of system states increases the effort needed to manually identify how system states correspond to components of the system's software.

[20] takes a software approach to detecting ESD interference in microcontrollers. They monitor the behavior of a phase-lock loop (PLL) embedded in the microcontroller; if the PLL unlocks, it can be assumed that the system has experienced an ESD shock. Their approach requires continuous polling of the PLL's status, adding a constant software overhead to the system. While it provides an excellent measure of ESD events on the microcontroller, it cannot measure peripheral ESD events because most peripherals do not contain a separate PLL that can be monitored by the microcontroller.

III. Approach Overview

Inducing ESD events on an embedded system peripheral causes bits to flip in its data or control lines. These flipped bits can lead to changes in register values, loss of synchronization between peripherals and the CPU, or data corruption. All of these effects are visible to software running on the embedded system and thus should be detectable by monitoring software.

Our work focuses on monitoring these effects with software that allows normal system operation along with monitoring. We primarily study changes in register values, as those values control the behavior of the peripheral device. In effect, each system state is represented by the n-tuple consisting of the values of each peripheral register at a specific time. Changes in register values represent changes in the peripheral's operating state. Some of these changes will be part of normal operation. When ESD is induced, however, we should observe new abnormal states or unexpected transitions between normal states. These abnormal states and transitions can indicate that the system is experiencing ESD.

IV. Software Monitoring

A. Initial Design

Our first design focused on directly reading USB register values from their physical memory addresses. We adapted and modified the Myregrw [21] software to better suit our needs as a softprobe for ESD. This software consists of a Linux kernel module and a program that communicates with it. The kernel module reads the values of requested physical memory addresses. The user-level program reads a configuration file specifying which addresses to request, repeatedly requests the data at those addresses, and stores that data to a file.

We configured Myregrw to record all of the values between the addresses 0x49000000 and 0x49000014. The data in this range are the control and status registers for the USB host interface. We injected ESD into the host controller while Myregrw continuously sampled the registers. In theory, ESD-caused changes should appear in the recorded register values.

However, the sampling rate of this softprobe was not sufficient to gather ESD induced errors. As with some of the works mentioned in [22], bottleneck issues prevented this methodology from performing reasonably. We empirically determined that the sampling rate of the software is, on average, 342 times per second. We can assume in the worst case that the system executes one instruction per cycle and would reset a register value after one instruction. The Mini2440 has a 400MHz clock. Thus $\frac{342}{400*10^6} * 100 = 0.000856\%$ would be the worst case probability of observing an error in the status register before it was changed. Considering this low probability and the lack of information recorded from our experiments, we devised a new measurement methodology with a higher sampling rate capable of recording additional register values.

A confounding issue with this approach is the competition for access to these values between the Myregrw driver and the USB host controller driver. By default, Linux drivers have execution priority over any user applications, meaning that it would be nearly impossible to read all of the register values after an error, but before the USB host controller driver modifies the registers. Therefore, in addition to providing a faster sampling rate, the new methodology must account for Linux drivers modifying the control and status register values.

B. Improved Design

In the improved approach, we modified the drivers for the USB host controller to enable logging of relevant data. The host controller driver consists of several functions that are called when certain events occur; for example, `ohci_irq` is called when an IRQ occurs for the host controller. Logging which function is being called gives a rudimentary idea of the operational state of the hardware.

We first enabled the debugging configuration already present in the driver. In addition, we configured the driver to log the register values along with the name of the function to the system log before the execution of each function in the driver. These modifications allow us to observe not only register state changes but also the order in which different driver functions are called. We consider this to be minimally invasive to the driver software since it only requires trivial modifications to the driver code and does not affect the logic of the driver itself.

V. Analysis Methodology

The log files generated by the instrumented driver consist of lines each having a timestamp, register name, and associated register value. We must first convert these log files into lists of system states defined by the n-tuple of values of all recorded registers at a specific time. Because a state is created each time the register values are recorded, each log will contain repeated states when the driver repeats an operation. We will need to find these equivalent states and combine them to derive a state transition graph. In addition, since certain register values change every time the system is restarted, we will perform deduplication to compare execution traces from different tests. The analysis code's operation can be summarized as follows:

1) Parse the log file to create states based on the registers' information.
2) Deduplicate these execution traces to derive a per-test execution graph.
3) Deduplicate the execution graphs of different tests to derive a universal execution graph.
4) Using this execution graph and each test's execution trace, perform statistical analysis on the testing data.

A. Unique States

The first stage in the analysis parses the log file for each simulation. We read the register values for each state from a log file. After creating tuples for each of the states in that file, we deduplicate the list of states to create a list of unique states. We then use the original list of states to record the order in which the unique states were reached. States that were executed sequentially show transitions between unique states; combined with the unique state graph, they form a state machine for the host controller's operation. For statistical analysis, we also record the number of times each transition is taken.

B. Globally Unique States

The next element of analysis combines the data from each log into a global state machine. The process is very similar to that for developing the unique state graph for each log. Certain registers for the host controller contain memory addresses that change every time the driver is reloaded. These registers are `HcPeriodCurrentED`, `HcBulkCurrentED`, `HcFmRemaining`, `HcHCCA`, `HcControlHeadED`, `HcControlCurrentED`, `HcBulkHeadED`, `HcFmNumber`, and `HcDoneHead`. Since these values will differ from test to test, they should not be considered when comparing states from different logs. However, changes in these values in a single test may indicate ESD; therefore, we create a new globally unique state each time the value changes.

C. Graph Analysis

We divide the data collected from test runs into two groups: baseline and ESD-exposed. Baseline logs are logs of the system operating normally and provide us with the system's expected state machine. ESD-exposed logs then show how the system transitions into and out of unexpected behavior due to ESD exposure.

After we create the graph of globally unique states, we can analyze the baseline and ESD-exposed logs individually to observe how system behavior differs between them. To gain an understanding on how the system varies between normal operation and operation with ESD interference, we can subtract the set of states reached in baseline logs from the set of states reached in ESD-exposed logs to get a list of states only reached during ESD injection. These state sets can be used to show where and when the system transitioned into a state that is very likely to have been caused by ESD. Similarly, we can determine which transitions between states are present only during ESD exposure.

VI. Test Hardware

The board used during the testing is the FriendlyArm Mini2440 embedded development board with a Samsung S3C2440 ARM9 processor [23]. The USB host interface conforms to the Open Host Controller Interface specifications [24]. The system ran a modified Linux kernel, based on the version 2.6.29 kernel downloaded from the FriendlyArm website [23]. We set up the system with our logging software and connected it to a PC to control it during the tests. During testing, a standard flash drive holding a large file was connected to the board's USB port. To ensure that the host controller is active during ESD injection, we copied that file to or from the flash drive during tests.

ESD interference was injected using electric (E) field and magnetic (H) field probes powered by a transmission line pulse (TLP) generator. For each probe, multiple tests were run with varying pulse voltages. In addition, different sizes of probes were used to adjust the intensity of the fields injected. The E-field does not have an orientation; we positioned it across the USB port or over the host controller IC. E-field interference was injected using an EZ-3 probe at voltages between 500 and 5500 volts with a pulse width between 0.1 and 0.25 seconds. Because the magnetic fields generated by the H-field probe are polarized, we conducted tests with the probe in parallel with and perpendicular to the data and control lines. We used two probes, the HX-5 and the HX-1T2, injecting ESD betweeen 500 and 8000 volts with pulse widths between 0.1 and 0.6 seconds. The board was more resilient to H-field interference, allowing us to perform H-field tests with more intense ESD conditions than were possible with E-field tests.

TABLE I. `HcInterruptEnable` AND `HcInterruptDisable` VALUE LIKELIHOOD

Value	**Baseline**	**ESD-exposed**		
		`Enable`	**`Disable`**	**Difference**
0x8000005e	0.22289	0.20041	0.20041	0
0x8000001a	0.01350	0.09677	0.09666	0.00011
0x8000005a	0.76156	0.65932	0.65943	-0.00011
0x8000001e	0.00202	0.04359	0.04359	0

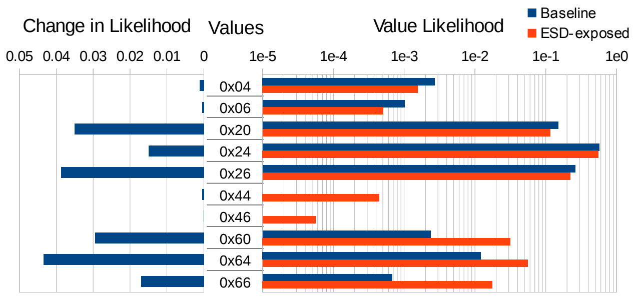

Fig. 1. `HcInterruptStatus` Register Values

VII. Results

A. Registers of Interest

Certain registers on the host controller can give indications of ESD. In particular, we consider the values of the registers for interrupt enabling and disabling (`HcInterruptEnable` and `HcInterruptDisable`), interrupt status (`HcInterruptStatus`), control (`HcControl`), and port status (`HcRhPortStatus0`).

`HcInterruptEnable` and `HcInterruptDisable` should be duplicates of each other when read. However, as shown in Table I, there are a few states in the ESD-exposed data where they are not duplicates. This may indicate ESD-induced bit flips or the system failing to properly update both registers when one is changed.

The `HcInterruptStatus` register values observed are shown in Figure 1 along with the likelihood of those values appearing in baseline and ESD-exposed logs and the absolute change in that likelihood due to ESD exposure. It shows a dramatic increase in values where the frame number counter overflowed (0x20,0x24,0x26,0x60,0x64,0x66) in the ESD-exposed logs, indicating that the system transmits many more frames during ESD exposure. In addition, values indicating the hub's status has changed (0x44,0x46,0x60,0x64,0x66) are also much more prevalent in ESD-exposed logs.

The `HcControl` register values provide a different aspect on the increase in the number of frames and hub status changes. Figure 2 shows a great increase in control frame processing (0x93) and a corresponding decrease in bulk data frame processing (0xa3). It is likely that the host controller's internal state is being corrupted by ESD exposure, resulting in an increase of status change frames. Data corruption in the bulk data frames would cause a decrease in the number of correctly transmitted frames and therefore less time overall being spent processing the correct data.

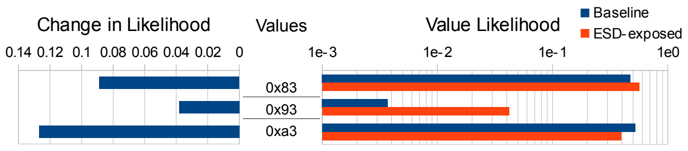

Fig. 2. `HcControl` Register Values

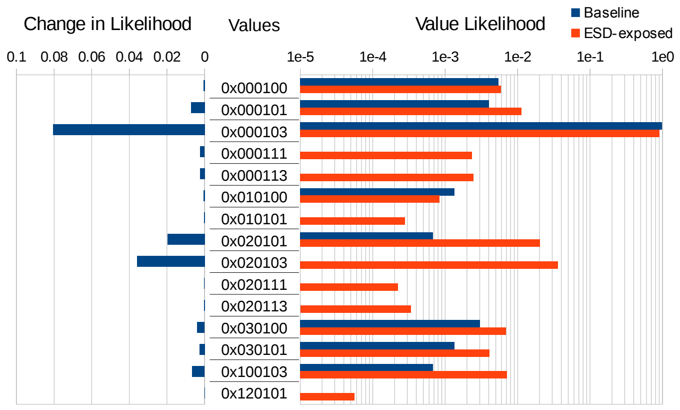

Fig. 3. `HcRhPortStatus0` Register Values

The `HcRhPortStatus0` register contains status information about the port the USB drive was plugged into during testing. Figure 3 shows recorded values, the likelihood of those values occurring in baseline and ESD-exposed logs, and the absolute effect of ESD exposure on the likelihood. We can see a marked decrease in normal operating states (0x100, 0x101, 0x103) and an increase in states indicating the port has been enabled or disabled (0x20103, 0x20111, 0x20113, 0x120101). As well, port resets (0x111, 0x113, 0x20111, 0x2113) were only observed in ESD-exposed logs. The prevalence of resets and toggling whether the port is enabled hint that the host controller is experiencing unexpected errors and attempting to recover by resetting the port's status. The presence of a port reset where the driver or host controller would not usually issue one is a particularly strong indicator of ESD exposure.

B. State Execution Graphs

Figure 4 shows a state graph of sample baseline and ESD-exposed logs. The set of nodes and solid arcs on the left of the figure is the state graph of the baseline log. The right of the figure consists of the additional states and transitions present in the sample ESD-exposed log. This state graph demonstrates several effects of ESD on the system state: ① transitions to abnormal states, ② transitions between abnormal states, ③ abnormal transitions between normal states, and ④ transitions from abnormal to normal states.

Consider how we should expect the system to behave under normal conditions and under ESD exposure. Normally, it should have a small number of common code paths and some edge case handling. Under ESD exposure, we should see a large number of different anomalous states caused by different register bits being flipped. Figure 5 shows the average number of occurrences per log file of states from baseline logs and ESD-exposed logs. We see a few normal states that are very common, with a small tail of less common states. There are far more unique states in ESD-exposed logs, and they are far less likely to occur. (We have omitted half of the ESD-exposed state tail to make the interesting portion of the graph more legible.) This graph provides a sanity check for our methodology; we can see that the data we have collected is behaving as expected.

Figure 6 compares the TLP pulse voltage with the percentage of transitions to or from states not in the baseline logs. The

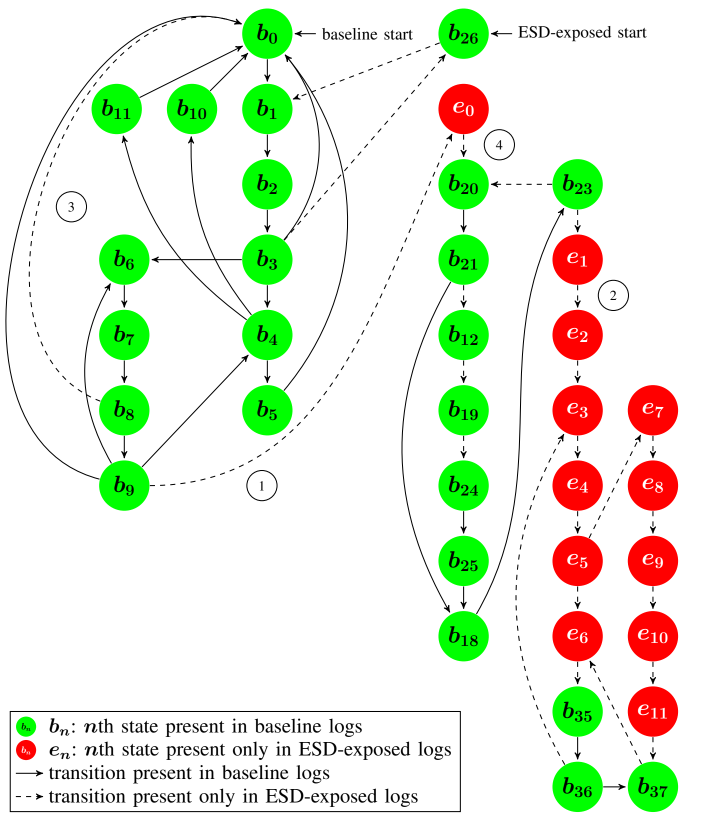
baseline start
ESD-exposed start
b_n: nth state present in baseline logs
e_n: nth state present only in ESD-exposed logs
transition present in baseline logs
transition present only in ESD-exposed logs

Fig. 4. State execution graph of one baseline log and one ESD-exposed log

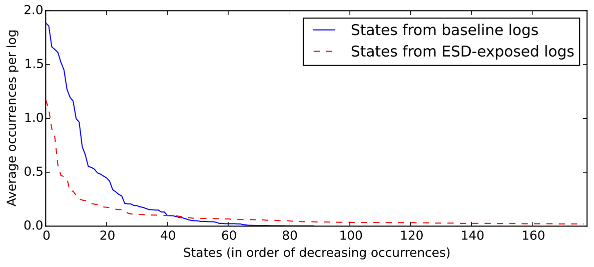

Fig. 5. Average State Occurrences Per Log

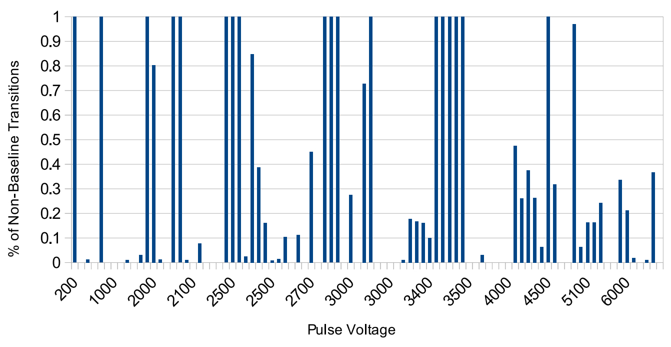

Fig. 6. Relationship between pulse voltage and ESD-caused transitions

lack of a clear relationship between observed ESD coupling and pulse voltage indicates that there are confounding factors between ESD exposure and system behavior. These factors may include field type and orientation, injection location, pulse frequency, and the operation being performed by the host controller at the time of injection. In addition, the ESD injection may cause the system to crash almost instantaneously, so the resulting state log will have relatively few states caused by ESD. More work is needed to assess the effect each of these factors has on system operation.

VIII. Classifier

So far, we have focused on determining how the system behaves when exposed to ESD. However, we may want to reverse this relationship and determine whether or not the system is being exposed to ESD based on its observed behavior. We present a preliminary finite state classifier for the host controller state data. While the approach here is limited to offline training and classification, it has been designed with the ability to be easily converted to real-time operation. As such, the classification process is computationally inexpensive in order to reduce overhead on an embedded system.

The finite state classifier itself has two states: ‘normal’ and ‘ESD-exposed’, representing the status of the system. Each state the host controller enters causes a transition between classifier states. The classifier has a concept of ‘certainty’ that describes the likelihood that the current classifier state is correct (e.g. the likelihood that the system is experiencing ESD when the classifier is in the ‘ESD-exposed’ state). Certainty is quantified as a value between -1 and 1, with -1 representing complete certainty that the classification is ‘normal’ and 1 representing complete certainty that the classification is ‘ESD-exposed’. An ESD event would thus appear as an increasingly positive certainty that the system is ESD-exposed. Recovery from such an event would appear as an increasingly negative certainty.

A. *Training*

In order to deduce when the system is exposed to ESD from our data, we must consider three classes of states: states appearing only in baseline logs, states only appearing in ESD-exposed logs, and states appearing in both. We say we are certain that states only in baseline logs indicate normal behavior, and states only in ESD-exposed logs are a certain indicator of ESD exposure. Certainty is implemented by assigning each system state a weight. Weights are selected to be within the range $[-1, 1]$ such that a positive weight value indicates an ESD-exposed state and a negative weight value indicates a normal state.

Weight value assignment must compensate for the fact that the training data will contain more ESD-exposed logs than baseline logs, since we performed more ESD-exposed tests. Weights are assigned as follows:

Let B and E be the sets of all base and error states, respectively.

Define constants for base states $w_b = -|E|$ and for ESD-exposed states $w_e = |B|$.

For a unique state i, let b_i be the number of times state i is present in B, and e_i be the number of times it is present in E.

For each unique state i, the normalized weight is defined as:

$$w_i = \frac{w_b * b_i + w_e * e_i}{|w_b * b_i| + |w_e * e_i|}$$

which can also be written as

$$w_i = \frac{-|E| * b_i + |B| * e_i}{|E| * b_i + |B| * e_i}$$

B. *Classification*

We can use these weights to build a classifier for the logs we have collected. Currently, the time window of data the classifier operates on is an entire log file of states, but it could be easily changed to a moving-window classifier for a real-time application.

The weights can be used to classify a log as follows: Let S_L be the set of unique states in log L, and s_i the number of times state i appears in the log. The log's classification, C_L, is then

$$C_L = \frac{\sum_i^{S_L} w_i * s_i}{|L|}$$

If C_L is positive, we classify the log as having experienced ESD, and if it is negative, we classify it as having not experienced ESD.

Because the training data for the classifier is not guaranteed to contain all states, certain states in the test data may not have assigned weights. We default these weights to 0, so they have no effect on the certainty of the classifier.

Classifier precision can be measured by the percentage of states for which weights are defined. We define accuracy by whether baseline logs are classified as normal operation and ESD-exposed logs are classified as ESD-exposed operation.

We tested this procedure over 22 base logs and 113 ESD-exposed logs. Classifier performance over k-fold verification with $k = 10$ was an 68.5% average accuracy and 84.6% average precision.

TABLE II. Classifier Performance for Different n-sitions

n	Avg. Accuracy	Avg. Precision	Avg. Table Entries
1	68.5%	84.6%	428
2	85.4%	62.5%	1256
3	84.6%	39.7%	2231
4	86.2%	22.7%	3324
5	87.3%	14.2%	4348
6	87.3%	8.3%	5258

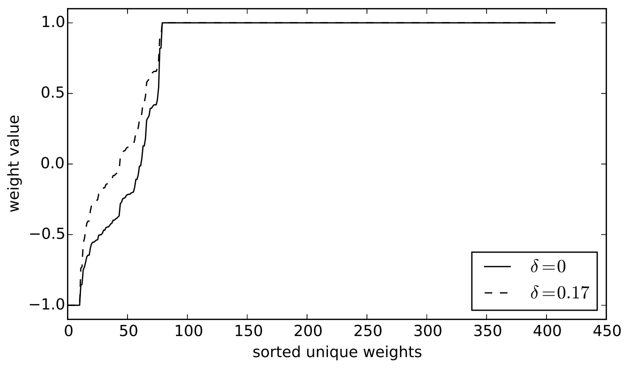

Fig. 7. Weights With and Without δ

TABLE III. Classifier Performance for Different n-sitions with δ

n	Avg. Accuracy	Avg. Precision	Best δ
1	88.5%	84.6%	0.17
2	88.5%	62.5%	0.275
3	86.9%	39.6%	0.138
4	86.9%	22.7%	0.115
5	87.3%	14.2%	0.125
6	88.9%	8.3%	0.377

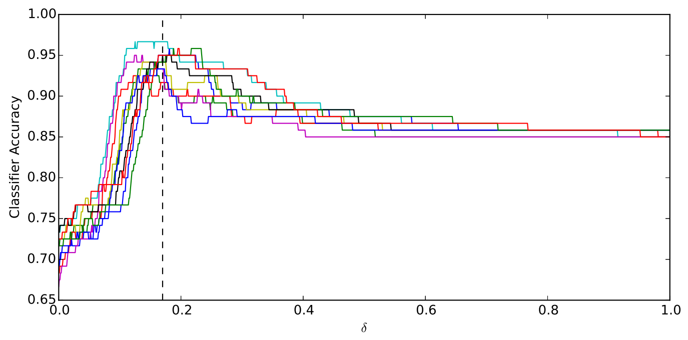

Fig. 8. Effect of δ on accuracy

C. System State Transitions

We hypothesize that the behavior of the host controller is non-Markov; that is, knowing how the system reached the current state may allow us to predict the states to which it can transition with higher accuracy. In addition, ESD exposure may cause the system to abnormally transition between two normal states. This effect of ESD is not captured in the current classifier design. We can provide some contextual knowledge for the classifier by classifying system transitions instead of system states. Additionally, we can extend the concept of a transiton, which is a 2-tuple of states, to an 'n-sition', an n-tuple of states in the order they were executed. In graph theoretic terms, an n-sition is a path of length $n-1$. One disadvantage of adding context in this fashion is the increasing likelihood of encountering an n-sition in the test data that was not present in the training data. Thus, we must choose a tradeoff between increasing accuracy and decreasing precision. Another disadvantage is the increased storage overhead for the weight table, which must be able to fit into memory on an embedded system. Table II shows classifier performance for different n-sitions, again averaged over $k = 10$ folds. The $n = 1$ data is the same as that presented in the previous section.

D. Delta

During testing, we determined that the classifier was performing poorly in part due to an underestimation of the weights of states indicating ESD exposure. We introduced the δ parameter as a means of adjusting the weight distribution.

Base and ESD-exposed constants are implemented as $w_b = -|E| - (|E| + |B|) * \delta$ and $w_e = |B| + (|E| + |B|) * \delta)$.

Figure 7 shows the change in the distribution of weights without δ and with $\delta = 0.17$.

Running $k = 10$-fold cross-validation for several n-sitions and values of δ between 0 and 1, the classifier performs as shown in Table III.

To demonstrate that δ does not cause the model to overfit, Figure 8 shows the effect of δ on the accuracy of the classifier for each fold. The accuracy shown is the accuracy of the classifier on the data it was trained on. It is essential that we choose a value of δ based only on information gained from the training data lest we bias the classifier towards the testing data. The vertical dashed line shows the value of δ that maximizes accuracy across all folds. If δ were causing the classifier to overfit to the training data, we would expect to see each fold having a wildly different optimal δ. However, in this plot, each fold's accuracy peaks near the same δ value, indicating that the model does not overfit.

As we expect, δ has no effect on precision, but it does have an effect on accuracy. It allows us to reduce the amount of context the classifier needs to perform well (smaller n), allowing us to have a more precise classifier without losing accuracy.

IX. Conclusion

We have presented a software-based methodology for detecting ESD events on embedded system peripherals. This methodology approximates the state of the peripheral by reading the registers it exposes to the CPU with an instrumented kernel driver for the peripheral.

We applied this methodology to a USB host controller on an embedded system running Linux. We demonstrated that we are able to observe states and transitions that the system experiences only when exposed to ESD.

The relationship between the recorded errors and ESD can be reversed. Doing so allows us to predict, based on the errors that the software experiences, when and where the system experiences ESD. We can apply this in several ways: components that have received ESD can be identified, either for replacement (if the goal of the experiment is to repair hardware that has experienced ESD) or for improvement (if the goal is to reduce the effects of ESD on a peripheral). In addition, software can be written to recover from error states in a more efficient and automatic fashion. Software may also be able to compensate for the effects of ESD, allowing operation to continue in hostile environments, although at the cost of reduced performance and more software overhead.

Our work contains a preliminary classifier that can be used to identify this relationship between system errors and ESD exposure. It is especially designed to provide a foundation for developing ESD-robust software. The next step beyond this research would be to implement a real-time classifier running on the embedded system. To reduce processing and memory overhead on the embedded system, the number of weight table entries may be able to be reduced with principal component analysis or other statistical analysis techniques.

Another avenue for research is correlating system states with ESD injection on a specific location on the board, which could give insight into which components have experienced ESD for performing repairs or assist circuit designers in shielding the board from particular error states. One could also study system states from a software perspective to determine how best to recover from certain ESD-induced errors. Finally, applying this methodology to other peripherals and embedded systems may lead to additional insights for software monitoring. In particular, applying this methodology in tandem with PCB schematic and chip layout analysis would provide a bridge between software-observed and hardware-observed ESD effects.

References

[1] Z. Li, J. Xiao, B. Seol, J. Lee, and D. Pommerenke, "Measurement methodology for establishing an IC ESD sensitivity database," in *Electromagnetic Compatibility (APEMC), 2010 Asia-Pacific Symposium on*, pp. 1051–1054, April 2010.

[2] D. P. J. Koo and G. Muchaidze, "Finding the root cause of an esd upset event," 2006.

[3] K. H. Kim and Y. Kim, "Systematic analysis methodology for mobile phone's electrostatic discharge soft failures," *Electromagnetic Compatibility, IEEE Transactions on*, vol. 53, pp. 611–618, Aug 2011.

[4] T. Schwingshackl, B. Orr, J. Willemen, W. Simburger, H. Gossner, W. Bosch, and D. Pommerenke, "Powered system-level conductive tlp probing method for esd/emi hard fail and soft fail threshold evaluation," in *Electrical Overstress/Electrostatic Discharge Symposium (EOS/ESD), 2013 35th*, pp. 1–8, Sept 2013.

[5] G. Muchaidze, J. Koo, Q. Cai, T. Li, L. Han, A. Martwick, K. Wang, J. Min, J. Drewniak, and D. Pommerenke, "Susceptibility scanning as a failure analysis tool for system-level electrostatic discharge ESD problems," *IEEE Transactions on Electromagnetic Compatibility*, vol. 50, pp. 268–276, May 2008.

[6] K. Wang, J. Koo, G. Muchaidze, and D. Pommerenke, "ESD susceptibility characterization of an EUT by using 3D ESD scanning system," in *International Symposium on Electromagnetic Compatibility EMC*, vol. 2, pp. 350–355, August 2005.

[7] S. Siha, H. Swaminathan, G. Kadamati, and C. Duvvury, "An automated tool for detecting ESD design errors," in *Electrical Overstress/Electrostatic Discharge Symposium Proceedings*, p. 208 217, October 1998.

[8] N. Monnereau, F. Caignet, D. Trémouilles, N. Nolhier, and M. Bafleur, "A system-level electrostatic-discharge-protection modeling methodology for time-domain analysis," *Electromagnetic Compatibility, IEEE Transactions on*, vol. 55, no. 1, pp. 45–57, 2013.

[9] C. Watterson and D. Heffernan, "Runtime verification and monitoring of embedded systems," *Software, IET*, vol. 1, no. 5, pp. 172–179, 2007.

[10] N. Delgado, A. Gates, and S. Roach, "A taxonomy and catalog of runtime software-fault monitoring tools," *IEEE Transactions on Software Engineering*, vol. 30, no. 12, pp. 859–872, 2004.

[11] B. Schroeder, "On-line monitoring: a tutorial," *Computer*, vol. 28, p. 72 78, June 1995.

[12] S. Choudhuri and T. Givargis, "Flashbox: a system for logging nondeterministic events in deployed embedded systems.," in *SAC* (S. Y. Shin and S. Ossowski, eds.), pp. 1676–1682, ACM, 2009.

[13] T. Reinbacher, M. Horauer, and A. Steininger, "A runtime verification unit for microcontrollers," in *System, Software SoC and Silicon Debug Conference (S4D)*, p. 1 6, September 2012.

[14] J. Wang, K. Sun, and A. Stavrou, "Hardware-assisted application integrity monitor," in *Hawaii International Conference on System Science (HICSS)*, pp. 5375 – 5383, January 2012.

[15] A.-C. Liu and R. Parthasarathi, "Hardware monitoring of a multiprocessor system," *IEEE Micro*, vol. 9, no. 5, p. 44 51, 1989.

[16] W. R. Deniston, "SIPE: A TSS/360 software measurement technique," in *Proceedings of the 1969 24th national conference*, ACM '69, (New York, NY, USA), pp. 229–245, ACM, 1969.

[17] J. Slye and E. Elnozahy, "Support for software interrupts in log-based rollback-recovery," *Computers, IEEE Transactions on*, vol. 47, no. 10, pp. 1113–1123, 1998.

[18] M. J. Pont and H. Ong, "Using watchdog timers to improve the reliability of TTCS embedded systems: Seven new patterns and a case study," in *Proceedings of VikingPLOP*, 2002.

[19] S. Elbaum and J. Munson, "Software black box: an alternative mechanism for failure analysis," in *Software Reliability Engineering, 2000. ISSRE 2000. Proceedings. 11th International Symposium on*, pp. 365–376, 2000.

[20] S.-Y. Yuan, Y.-L. Wu, R. Perdriau, and S.-S. Liao, "Detection of electromagnetic interference in microcontrollers using the instability of an embedded phase-lock loop," *IEEE Transactions on Electromagnetic Compatibility*, vol. PP (early access preprint), no. 99, pp. 1–8, 2013.

[21] "edwinrong/myregrw." https://github.com/edwinrong/myregrw. Accessed: 2015-01-23.

[22] S. Callanan, D. Dean, M. Gorbovitski, R. Grosu, J. Seyster, S. Smolka, S. Stoller, and E. Zadok, "Software monitoring with bounded overhead," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pp. 1–8, 2008.

[23] "Downloads - friendlyarm." Accessed: 27/02/2013.

[24] "1394 Open Host Controller Interface Specification - Microsoft." http://www.microsoft.com/whdc/system/bus/1394/OHCI.mspx. Accessed: 2015-01-23.

[25] N. Jarus, A. Sabatini, P. Maheshwari, and S. Sedigh Sarvestani, "USB kernel driver for ESD instrumentation." https://github.com/sendecomp/esd-kernel-drivers, 2015.

[26] N. Jarus, A. Sabatini, P. Maheshwari, and S. Sedigh Sarvestani, "ESD data analysis scripts." https://github.com/sendecomp/esd-analysis, 2015.

[27] N. Jarus, A. Sabatini, P. Maheshwari, and S. Sedigh Sarvestani, "USB host controller ESD exposure dataset." https://github.com/sendecomp/esd-data, 2015.