Hardware-based Probabilistic Threat Detection and Estimation for Embedded Systems

Nadir A. Carreón
Department of Electrical and Computer
Engineering
University of Arizona
Tucson, Arizona, USA
nadir@email.arizona.edu

Sixing Lu
Department of Electrical and Computer
Engineering
University of Arizona
Tucson, Arizona, USA
sixinglu@email.arizona.edu

Roman Lysecky
Department of Electrical and Computer
Engineering
University of Arizona
Tucson, Arizona, USA
rlysecky@ece.arizona.edu

Abstract—With billions of networked connected embedded systems, the security historically provided by the isolation of embedded systems is no longer sufficient. Both proactive security measures that prevent intrusions and reactive measures that detect intrusions are essential. Anomaly-based detection is a common reactive approach employed to detect malware that has evaded proactive defenses by observing anomalous deviations in the system execution. Timing-based anomaly detection detects malware by monitoring the system’s internal timing, which offers unique protection against mimicry malware compared to sequence-based anomaly detection. However, previous timing-based anomaly detection methods focus on each operation independently at the granularity of tasks, function calls, system calls, or basic blocks. These approaches neither consider the entire software execution path nor provide a quantitative estimate of the presence of malware. This paper presents a novel model for specifying the normal timing for execution paths in software applications using cumulative distribution functions of timing data in sliding execution windows. We present a probabilistic formulation for estimating the presence of malware for individual operations and sequences of operations within the paths, and we define thresholds to minimize false positives based on training data. Experimental results with a smart connected pacemaker and three sophisticated mimicry malware demonstrate improved performance and accuracy compared to state-of-the-art timing-based malware detection.

Keywords—Timing-based threat detection; Anomaly detection; Embedded system security; Non-intrusive hardware

I. INTRODUCTION

Internet connected devices have grown explosively and are estimated to reach several billions by 2020 [1]. At the same time, more than 100 million new malware were created in 2016 [2], which is indicative of the scope of the potential threat to the security and privacy of users and companies. Embedded systems used to be physically isolated (e.g., the computer inside a locked automobile), which in turn provided some measure of security because an attacker had to access to the system physically to perform an attack. However, an increasing number of previously secured embedded systems are now connected to the Internet, which opens the door to attackers. While Internet connectivity has tremendous benefits across wide ranging applications, including automobiles, medical devices, etc., that connectivity brings new threats to these systems. The impacts of malware can range from minor inconveniences to life-threatening situations. For example, Li et al. [3] demonstrated that by using publicly

available information, a malicious packet could be transmitted to an insulin pump that would deliver a fatal dose of insulin.

While proactive approaches (e.g., secure communication protocols, static application security testing) are essential, runtime intrusion and malware detection are also needed to detect when attackers are able to circumvent a system’s defenses. Malware detection can be broadly categorized into signature-based detection and anomaly-based detection. Signature-based detection detects malware by matching execution behaviors, code patterns, etc. to a library of known malware. This requires storing a large library for the growing number of malware and their variants, and limits the ability to detect zero-day exploits [4]. In contrast, anomaly-based detection detects malware by detecting deviations in execution behaviors at runtime from a pre-defined model of normal system behaviors. Anomaly-based detection commonly focuses on monitoring the internal sequence of operations within the system [5][6][7], where any deviation from the expected sequence would be considered anomalous. However, sequence-based anomaly detection does not protect against mimicry attacks. Wagner et al. [8][9] evaluated several sequence-based anomaly detection methods and demonstrated that malware can hide their presence by mimicking the correct execution sequence of the target application/device. This sophisticated type of malware is known as *mimicry malware*.

Timing-based anomaly detection improves the detection rate and resilience to mimicry malware by adding timing information to the normal system model. Time is a critical component in embedded systems and strict timing constraints are often required to ensure system correctness and safety. The resulting time sensitivity means that small changes in the timing of some operations can adversely affect the system execution, in the worst case leading to system failures. By monitoring both the internal timing of operations and the execution sequence, timing-based anomaly detection can detect mimicry attacks by observing the resulting changes in system timing. Several approaches [10][11][12] use timing of individual operations to detect malware at runtime, but these approaches often suffer from high false positive rates, which is a well-known problem in existing anomaly detectors.

In this paper, we present a statistical approach for modeling the normal system behavior of embedded applications using cumulative distribution functions (CDF) of timing data within sliding execution windows. Instead of independently focusing on

single operations, the normal timing model enables the monitoring of each execution path within a software application. For each execution path, a probabilistic formulation is used to estimate the presence of malware for individual operations and sequences of operations. To ensure rapid detection, a hardware-based runtime detector analyzes timing samples within a fixed-size sliding window, comparing the CDFs of the sliding window against the normal system model. The detector calculates an estimated probability of malware by measuring the percentage of the CDF that falls outside the normal boundaries for each operation and for the entire path. The malware detector is implemented in hardware that interfaces to a processor's trace port, which enable the runtime detection to be performed without impacting the software execution. Experiments with a smart connected pacemaker prototype and three mimicry malware were conducted to evaluate the detection rate, false positive rate, detection latency, area, and power consumption of the presented approach. We further compare our method to a state-of-the-art timing-based malware detection.

II. RELATED WORK

Several efforts have been made to develop timing-based anomaly detection. Most of these existing approaches detect the timing of single operations and do not consider the history of execution timing at runtime. Also, some approaches incur high false positive rates. These approaches primarily focus on detection accuracy and detection rates, which are both critical aspects of malware detection, and many achieve excellent results. But none specifically consider minimizing the false positive rate or monitoring complete execution paths. In contrast, our presented approach attempts to minimize the false positive rate, maintain a high detection rate, consider complete execution paths, and provide a quantitative estimate of the probability of malware.

Patel et al. [13][14] presented a MPSoC intrusion detection method that monitors the execution's sequence and time of basic blocks within the software application using a control flow map. The detection method inserts custom instructions in the software binary to identify basic blocks. Those instructions transmit the execution data to a dedicated processor for analysis. The overhead of the instrumented code and additional processors incur a performance overhead between 6.6% and 44% and an area overhead of 26.9%. For many embedded systems, such as implantable medical devices, the impact on the performance of the software application is often infeasible or prohibitively expensive.

Zimmer et al. [10] proposed a timing-based intrusion detection system at the granularity of function calls and return paths, specifically designed to detect code injection attacks. However, this approach makes decisions based on individual operations, instead of considering entire execution paths. This can lead to a high number of false positives.

Lu et al. [12] developed a range-based runtime anomaly detection method that monitors the best-case execution time (BCET) and worst case execution time (WCET) of individual operations within a software application. Although this approach can detect malware efficiently, it independently makes decisions based on individual operation without considering statistical characteristics of execution timing. By only considering the timing range defined the BCET and WCET of individual

events, the timing variability therein presents challenges in accurately training the model to achieve low false positives.

Yoon et al. [11] presented SecureCore, which monitors the timing distribution of basic blocks. At runtime, SecureCore measures the execution time of a basic block and estimates the probability of that time from a statically determined timing distribution. If that probability is less than a specified threshold (e.g., 5%), SecureCore flags the execution as potentially malicious. This approach is highly dependent on the threshold utilized, since false alarms triggered at runtime may decrease the overall system performance. As such, a balanced threshold is critical for the correct functionality of the system. However, it should be noted that false positives are inherent in this approach, and when applied to operations at a coarse granularity (e.g., system/function calls) the false positive rate can be very high.

III. ASSUMPTIONS AND THREAT MODEL

The goal of CDF-based anomaly detection is to detect sophisticated mimicry malware with minimum or no false positives given the following assumptions and threat model.

1. The target malware is mimicry malware [8][9], which attempts to evade anomaly detection by mimicking normal execution behavior. Mimicry malware interleaves malicious and normal execution and are sophisticated enough to escape detection from simple sequence-based anomaly detection. Sequence-based anomaly detection is also used in our approach, which can detect non-mimicry malware and necessitates an attacker's need to use mimicry malware.
2. The attacker either has access to system software or can simulate the system execution to determine the system's execution sequence, which is needed to create mimicry malware. The attacker is able to remotely insert the malware into the system utilizing software that exploits a vulnerability, which may be unknown or known but unpatched at the time of insertion. The anomaly-based malware detection presented in this paper focuses on detecting malware that has already been inserted in the system and not on detecting the software or system vulnerabilities that lead to the intrusion, both which are beyond the scope of this paper.
3. The target embedded application consists of multiple software tasks (or threads) executing on a single processor core, although we note that the detection method presented herein can be applied to other application scenarios including multicore systems.
4. The granularity of detection in this paper is at the level of system and function calls, generically called operations. The proposed approach can be applied at coarser or finer granularities following the same design principles.
5. The detection method considers individual execution paths, where a path is defined as a specific sequence of operations within the control flow of each software task.

To evaluate the CDF-based anomaly detection method, we consider three mimicry malware based on known malware (albeit from different applications) [15][16]. The File Manipulation malware breaks confidentiality and integrity by intruding into the file system and performing reads/writes on a target file. The

malicious goal varies between malware and target application (e.g., in the pacemaker application, modifying the records in cardiac log file to deceive the physician). The second is a Fuzz malware [16] that is commonly used to interfere with the system's pre-defined functionality by fuzzing (or slightly changing) data. Fuzz malware can be implemented in various levels, which enables the evaluation of the effectiveness of malware detection for different fuzzification levels. The Information Leakage malware reads the patient's cardiac activity log and transmits the data to a third-party server. These three malware were implemented by interleaving malicious operations with mimicked normal operations that overall follow the normal execution sequences and execution paths. Our primary threat is malware affecting legitimate executables, specifically mimicry malware, which assume an attacker knows which operations are monitored. As the approach detects deviations in execution sequences and timing, it can also indirectly detect other malware types.

IV. CDF-BASED ANOMALY DETECTION

Fig. 1 presents the design flow of the CDF-based anomaly detection method. The software application is first statically analyzed to determine the operations, o_i , and execution paths, p_i , within all software tasks. For each operation, the system is executed to collect training data by executing the system under various normal execution scenarios, each for a sufficient duration. We utilize the processor's trace interface [17] to observe the timing of operations without affecting the execution or timing thereof. For a specific window size and stride, the CDF analysis determines the CDFs per window within the training data. These CDFs are used to calculate the upper bound, $Bound_{upper}(o_i)$, and lower bound, $Bound_{lower}(o_i)$, of the CDF per operation, which can be used to detect deviations at runtime.

This approach is based on the Kolmogorov–Smirnov test (K-S test) [22] and seeks to detect anomalies by statistically comparing the distribution of timing data between a normal system execution model and the runtime execution. Without assuming a specific distribution, the K-S test can be used to test a sample with a reference distribution for equality, where the reference distribution is the normal system model and the sample is the runtime execution. To test if the sample and normal distribution are equal, the K-S test computes the CDF for both, and measures the maximum difference between the two. While the K-S test can be directly applied to detect anomalous executions, one would need to collect 1000s of timing samples for each operation before testing the distribution for equality, thus leading to unacceptably long detection latencies. Based on the K-S test, our approach measures the execution timing with a small execution window. To construct the normal system model, the CDF for each execution window in the training data is determined. However, instead of storing all CDFs, which would require prohibitively larger memory requirements, only the minimum and maximum boundaries of the CDFs across all windows are stored. At runtime, instead of calculating the maximum difference between two distribution, we calculate the percentage of samples that do not fall within these CDF bounds.

After determining the CDF boundaries, additional normal training data is used to calculate the maximum deviation the CDFs can have from the normal execution while still being considered normal, defined as the threshold. For each operation, the

Fig. 1. Design flow of the CDF-based anomaly detection. The CDF Analysis (per path) module is utilized both at runtime and in static analysis, with the only distinction being the input data will be extracted at runtime, which may or may not be malicious.

percentage of CDF values for each execution window outside the operation's CDF boundaries is used to determine an estimate of the probability of malware. False positives can be quantified by analyzing the estimated probability of malware for normal execution timing. The probabilities of malware for all operations within an execution path are analyzed to determine a per-path threshold, which defines a per-path probability beyond which the approach is confident about the presence of malware.

At runtime, the detector utilizes the CDF boundaries to estimate the probability of malware for each execution path. Timing samples are collected using the same window size and stride. The CDFs for each operation are calculated, and the percentage of CDF values outside the statically determined CDF bounds are used to estimate the probability of malware for each operation. For each execution path, the detection method calculates the probability of malware affecting the timing of the path's operations. This estimated probability is compared against the predefined threshold to determine if malware is present in the system.

Fig. 2. Conceptual overview of window-based CDF calculation, highlighting the sliding windows of timing samples used to calculate the CDF of an operation. The plot shows the CDFs for four different windows, from which the upper and lower CDF boundaries can be calculated.

A. Window-based CDF Boundary Construction

The CDF represents the distribution of timing samples within an execution window. Creating a model of the normal system behavior using CDFs allows one to estimate the percentage of overlap between runtime CDFs and the normal system model's CDFs. Additionally, it reduces the storage requirements compared to other approaches (e.g., KDE estimation).

Storing and analyzing the entire execution history for an operation is infeasible and would lead to prohibitively long detection delays. Therefore, the CDF-based anomaly detection collects and analyzes an operation's timing within a fixed execution window that maintains multiple execution timing samples. The window size, defined as the number of timing samples maintained, should be large enough for statistical analysis but small enough to reduce the detection delay. For each window, the stride defines how many new timing samples are collected before re-calculating the CDF. A smaller stride produces smaller changes in the CDF, but requires re-calculating the CDFs more frequently to detect malware. However, a larger stride would allow malware to execute longer before being detected, which could be fatal for some systems. Fig. 2 presents a conceptual overview of the window-based CDF calculations showing the resulting CDFs for four different execution windows and the resulting boundaries. In this example, the window size is 20 and the stride is 5. Thus, each CDF involves 20 samples, in which 25% are new timing samples and 75% are prior samples.

After the CDFs for all windows of an operation are obtained during the training stage, the boundaries that define the normal system's model can be determined. The red bolded lines in Fig. 2 illustrate the CDF bounds for the sample windows. The lower

Fig. 3. Three runtime CDFs plotted with the upper and lower CDF bounds (red solid lines), showing the overlap used to calculate the estimated probability of malware. Normal execution of the system (blue circles) is always inside the previously obtained boundaries. The CDFs for malware (black crosses and green triangles) results in a CDF that does not fall within the CDF boundaries.

boundary is constructed by points in the CDFs that have the lowest cumulative probability at each timing value, and the upper boundary is constructed by the points in the CDFs that have the highest cumulative probability at each timing value. These boundaries are configured into the anomaly detector and used at runtime. Instead of fitting the boundary curve to be a high dimensional representation, because a fixed window size is used, the CDF's cumulative probability will be discretized with a step size equal to the inverse of the window size. Thus, the CDF boundaries are stored as two scalar arrays, $Bound_{upper}(o_i)[\]$ and $Bound_{lower}(o_i)[\]$, that contain the timing values corresponding to each discrete cumulative probability step.

B. Estimating Probability of Malware and Threshold-based Malware Classification

For an operation o_i , the estimated probability of malware, $P_{estO_i}(M)$, depends on the percentage of CDF values outside the CDF boundaries defined in the normal system model. Fig. 3 presents an example demonstrating how the probability of malware is calculated. The red solid lines are the normal boundaries, and the dashed lines are the CDFs obtained from three different windows of runtime timing data. The black (crosses) CDF is completely outside the CDF boundary, and thus is estimated to have 100% malicious execution. In contrast, the blue (circles) CDF is completely within the CDF boundaries and thus is estimated to have 0% malicious execution. For a CDF that partially overlaps with the CDF boundary, the probability of malware is estimated as the percentage of points within the CDF that fall outside the boundaries. For example, the green (triangles) CDF has a probability of malware $P_{estO_i}(M) = 1 - (0.65 - 0.20) = 0.55$, which indicates there is estimated to be a 55% probability the execution is malicious. In practice, with the $Bound_{upper}(o_i)[\]$ and $Bound_{lower}(o_i)[\]$ arrays, the $P_{estO_i}(M)$ is calculated by determining the number of samples that fall outside these bounds. For example, if 19 of 20 timing values in a window are outside the CDF boundary, the estimated probability of malware $P_{estO_i}(M) = 0.95$.

Fig. 4. Annotated execution paths within a software task of a network-connected pacemaker. The left path has four monitored operations, and the right path has nine. Annotations next to each operation indicate the estimated probabilities of malware for that operation. The resulting estimated probabilities of malware for each execution path and shown below each path.

Individual operations are considered malicious if the estimated probability of malware, $P_{est}O_i(M)$ is greater than a predefined threshold. Instead of making a decision based on a single operation, which may yield high false positive rates, a more robust decision is made by considering the $P_{est}O_i(M)$ for multiple operations in an execution path. An execution path is a sequence of operations within a software task. The probability of malware execution in a path would be higher if the probability of more operations within the path are estimated to be malicious. The probability of malware for a path p_j is:

$$P_{est}p_j(M) = 1 - \prod_{i=0}^n (1 - P_{est}O_i(M)) \quad (1)$$

Fig. 4 presents two different example execution paths for the smart connected pacemaker application considered in this paper (see Section VI-A). For the execution scenario shown on the left, four operations are monitored and the estimated probability of malware for path is calculated as: $P_{est}p_1(M) = 1 - (1 - 0.07)(1 - 0.08)(1 - 0.10)(1 - 0.08) = 0.2915$. This indicates there is a 29.15% chance that the execution of the task along this execution path for the current execution window is malicious. The execution path on the right represents a different execution path in the same task, in which nine operations are monitored. The estimated probability of malware for this alternative path is: $P_{est}p_2(M) = 1 - (1 - 0.00)(1 - 0.01)(1 - 0.00)(1 - 0.01)(1 - 0.03)(1 - 0.00)(1 - 0.02)(1 - 0.00)(1 - 0.00) = 0.0683$, which indicates there is a only 6.83% chance this path's execution is malicious.

Whether malware is affecting the system execution is decided per execution path, comparing the path's estimated probability of malware to a path specific threshold. The threshold is

Fig. 5. Timeline of estimated probability of malware for both normal and malware (File Manipulation) execution highlighting the detection latency.

defined by the maximum probability of malware execution $P_{max}O_i(M)$ per operation. The $P_{max}O_i(M)$ is calculated by processing a second set of normal training data utilizing the same approach as in Section IV-B. The threshold for path p_j is:

$$Tp_j = 1 - \prod_{i=0}^n (1 - P_{max}O_i(M)) \quad (2)$$

Due to limitations of design-time training, some normal system executions may deviate from the CDF boundaries. Without accounting for these deviations, a high false positive rate would be expected. The path-based threshold is utilized to minimize that false positive rate. Equation (2) utilizes the minimum overlap found in the second training data set (normal data only), while equation (1) is utilized at runtime to obtain the estimated probability of malware for the entire path. For example, assume the minimum overlap throughout all windows of operation O_i for the second set of normal timing data is 0.90. This means that the highest estimated probability of malware for normal system execution is 0.10, which in turns means that a runtime estimated probability of malware greater than 0.10 will be reported as malware. If five operations within that path are monitored, and each has the same minimum probability, the path threshold $Tp_j = 1 - (1 - 0.10)^5 = 0.40951$. This approach strives to ensure the CDF-based anomaly detection is accurate with minimal false positives. It can also be observed that as the number of monitored operations increases, the threshold decreases, but the strictness of the approach remains.

C. Runtime Detection

At runtime, the threshold of each path and normal CDF boundaries are configured within the hardware-based malware detector. The malware detector collects timing samples of each operation by analyzing the signals from the processor trace port. Whenever the stride is reached for an operation's window, the detector calculates the CDF and $P_{est}O_i(M)$ for the operation. When the CDFs of all monitored operations within a path j are calculated, the anomaly detector calculates $P_{est}p_j(M)$ and compares that estimated probability with the threshold Tp_j . If $P_{est}p_j(M) > Tp_j$, the detector asserts a non-maskable interrupt indicating the presence of malware.

Fig. 6. CDF-based anomaly detection hardware architecture.

Detection latency is a critical indicator of the detection performance and is defined as the time between the moment when malware begins to execute and the moment the detector detects the anomalous execution and asserts the alert. For the CDF-based anomaly detection, the detection latency is primarily due to the window size and stride. Fig. 5 presents a demonstration of how the sliding window approach affects detection latency for a single execution path. As the malware begins to execute, the estimated probability of malware increases. With each stride, the sliding window contains more timing samples from the malware, which increases the estimated probability of malware. Once the estimated probability of malware exceeds the threshold, the detector asserts the presence of malware.

V. CDF-BASED ANOMALY DETECTION HARDWARE

Fig. 6 presents an overview of the CDF-based anomaly detection (CDFAD) hardware. The CDFAD consists of three main components, the *Operation Identification & Timing*, *Execution Window Update*, and *Malware Estimation & Threshold Analysis* components. Additionally, the CDFAD uses three local memories to store operations' configuration, execution windows, and estimated malware probabilities, respectively. The *Configuration Memory* stores the monitored operations' start and end addresses, CDF boundaries, and thresholds. The *Execution Window Memory* stores the measured timing samples for the current execution window for all operations, as well as the number of new timing samples measured in the current stride. The *Malware Estimation & Threshold Analysis Memory* stores the current estimate of the probability of malware for all operations.

The *Operation Identification & Timing* component directly interfaces with the microprocessors' trace port to detect the execution of the monitored operations, using the operation's PC addresses. When the address for an operation's start address is detected, an internal timer is started to measure the execution for the current operation execution. That timer is then stopped when the operation's end address is detected. The *Execution Window Update* component receives the current measured timing value and operation ID from the *Operation Identification & Timing* and updates the corresponding entry within the *Execution Window Memory*. If the stride is reached, a window stride event signal is asserted to signal the *Malware Estimation & Threshold Analysis* component to perform the CDF-based analysis for this

operation. The *Malware Estimation & Threshold Analysis* component reads the current execution window from the *Execution Window Memory* and the CDF boundaries and threshold from the *Configuration Memory*, calculates the $P_{estOp_i}(M)$ of the current execution window, storing the results in the *Estimated Malware Probability Memory* component. If $P_{estOp_i}(M)$ is greater than the threshold, the CDFAD asserts the non-maskable interrupt.

We synthesized the CDFAD hardware targeting an Artix-7 XC7A200T FPGA with Vivado 2016.4. The synthesized hardware support 32 operations, using 32-bit registers for both the operation's addresses and timers, and using block RAMs (BRAMs) to implement the local memories. The CDFAD hardware requires 6,481 lookup tables (LUTs), 7,666 flip-flops (FF), and three BRAMs (2x32Kb, 1x64Kb). No additional external memory or storage is needed. The CDFAD hardware has a maximum operating frequency of 128 MHz, which is sufficient for the target integration with a 100 MHz MicroBlaze based system. The CDFAD hardware has a peak power consumption of 66 mW and average power consumption of 41.9 mW, which corresponds to a power overhead of only 3.56%

VI. EXPERIMENTAL RESULTS

A. Smart Connected Pacemaker

We developed a smart connected pacemaker and implemented a complete system prototype using the Artix-7 XC7A200T FPGA. The pacemaker prototype is representative of a complex embedded system that monitors, analyzes, stores, and transmits data, while providing life and safety critical operations. The pacemaker, shown in Fig. 7, includes a simulated heart, a tremor sensor, an impulse pacer, and four timers. The simulated heart beats irregularly and reacts to the impulse pacer controlled by the pacemaker's software. The cardiac activity sensor interfaces to the simulated heart and sends the measured activity to the microprocessor via interrupts. The output from the cardiac activity sensor also controls the Atrio-Ventricular Interval (AVI) and the Ventricular-Atrial Interval (VAI) timers. These timers are used to maintain the appropriate delay between the atrial/ventricular and ventricular/atrial activation and will generate an interrupt if the AVI/VAI exceeds a specific interval configured by a physician. The PVARP/VRP timers filter noise in the ventricular and atrial channels, respectively [18][19].

Fig. 7. Smart connected pacemaker system architecture.

The pacemaker's software, which executes on a MicroBlaze processor, consists of three tasks and four interrupt service routines (ISRs). The ISRs interact with the pacemaker's cardiac activity sensor and timers. ISR operations include performing the atrial and ventricular pacing and recording ventricular and atrial activity. The first software task calculates the Upper Rate Interval (URI) and records cardiac activity to a daily log file. A second software task analyzes the cardiac activity and detects a high URI, which indicates the pacemaker cannot pace the heart correctly or that pacemaker's cardiac activity sensor has malfunctioned. In the operation of a high URI, the pacemaker immediately transmits a warning message to alert the physician. The third software task is responsible for communication, by which the physician can configure the pacemaker's settings, or a home monitoring device can access daily logs of the cardiac activity.

Finally, the CDFAD hardware was integrated within the system prototype, monitoring the signals of the MicroBlaze processor's trace interface [20] to non-intrusively observe and analyze the system execution at runtime to detect anomalies.

B. Detection Rate and False Positive Rate

Using the four mimicry malware, the detection and the false positive rates of the CDF-based anomaly detection are evaluated. The true positive rate (TPR) (i.e., detection rate) is calculated as the number of malware executions classified as malware, divided by the total number of malware executions.

$$TPR = \frac{\text{Malware executions classified as malware}}{\text{Total malware executions}} \quad (3)$$

Using a separate set of data, the false positive rate (FPR) is calculated as the number of normal executions classified as malware divided by the total number of normal executions.

$$FPR = \frac{\text{Normal executions classified as malware}}{\text{Total normal executions}} \quad (4)$$

Table I presents the FPR for each execution path affected by the mimicry malware, the overall FPR across all paths, and the average TPR (i.e., detection rate) for the CDF-based anomaly detection compared to a range-based (RB) anomaly detection method [12]. The range-based method detects anomalies by determining when the timing of monitored operations falls outside

TABLE I. FALSE POSITIVE RATES AND TRUE POSITIVES RATES FOR CDF-BASED ANOMALY DETECTION AND RANGE-BASED (RB) DETECTION [12].

Detection Method	FPR (%)				Overall FPR (%)	Avg. TPR (%)
	P 1	P 2	P 3	P 4		
CDF	0.00	0.00	0.00	0.00	0.00	96.2
RB [12]	0.04	0.55	2.41	0.00	2.41	83.8

TABLE II. PER-PATH TPR FOR CDF-BASED ANOMALY DETECTION AND RANGE-BASED (RB) DETECTION [12].

Detection Method	TPR (%)		
	Path	CDF	RB [12]
File Manipulation	P1	99.8	2.1
	P2	94.4	83.7
	P3	99.5	100.0
Fuzz 20%	P1	99.5	90.3
	P2	80.0	100.0
	P3	99.5	100.0
Fuzz 100%	P1	99.0	62.9
	P2	91.3	100.0
	P3	99.5	100.0
Information Leakage	P4	99.0	99.0
Average		96.2	83.8

of a [WCET, BCET] range. While the ranges for operations can be either calculated using one of the many analysis techniques [21], we used experimentally collected normal training data to construct the CDF-based model to determine the ranges, following the same approach proposed in [12], ensuring a fair comparison. Additionally, we monitored the same operations within the same execution paths for the smart connected pacemaker for both approaches. Overall, the CDF-based detection achieves a 12.4% higher TPR compared to the range-based detection. The CDF-based detection also achieves a per-path and overall FPR of 0%, indicating the threshold determined during training effectively minimizes the FPR. In contrast, the FPR for the range-based approach is as high as 2.4%.

Table II presents the per-path TPR for each malware and the average TPR across all malware for the two detection methods. For Path 1, Path 3, and Path 4, the CDF-based anomaly detection has a TPR greater than 99% for all malware. For Path 2, the CDF-based anomaly detection achieves a TPR greater than 91% for the File Manipulation and Fuzz 100% malware. For the Fuzz 20% malware, which exhibits the smallest deviation in execution timing and is thus the hardest to detect, the approach achieves a TPR of 80%. For the File Manipulation malware, the CDF-based detection approach yields a significant increase in the TPR compared to the range-based approach for Path 1, with an increase of 97.7%. Across all malware considered, the CDF-based approach has a higher TPR for Path 1. On the other hand, for Path 2 and Path 3, the range-based detection achieves a higher TPR for the two Fuzz malware, but again at the expense of an increase in the FPR. Overall, the CDF-based approach achieves a higher average TPR and lower FPR (i.e., 0% FPR).

C. Analysis of Design Tradeoffs

Within the presented approach, we determined the window size of 20 at design-time to minimize the FPR and minimize hardware requirements. However, the window size also affects

TABLE III. ANALYSIS OF PER-PATH TPR, AVERAGE FPR, AND DETECTION LATENCY FOR DIFFERENT WINDOW SIZE CONFIGURATIONS.

Window Size	TPR (%)										FPR (%)	Detection Latency (Cycles)
	File Manipulation			Fuzz 20%			Fuzz 100%			Information Leakage		
	P1	P2	P3	P1	P2	P3	P1	P2	P3	P4		
20	99.8	94.4	99.5	99.5	80.0	99.5	99.0	91.3	99.5	99.0	0.0	668
40	99.7	92.6	99.0	98.0	60.0	99.0	98.0	87.0	99.0	99.0	0.0	1192
60	99.6	88.9	99.0	97.0	60.0	99.0	97.5	82.6	99.0	97.5	0.0	1716
80	99.5	87.0	99.0	96.5	40.0	99.0	97.0	78.3	99.0	96.5	0.0	2240
100	99.4	81.5	99.5	95.5	30.0	99.0	95.5	73.9	99.0	98.0	0.0	2764
120	99.4	79.6	99.5	94.5	20.0	98.5	95.0	73.9	98.5	96.0	0.0	3288
140	99.4	77.8	99.5	94.0	20.0	98.5	94.5	73.9	98.0	94.0	0.0	3812
160	99.3	77.8	100.0	93.5	20.0	98.5	94.5	69.6	99.0	93.5	0.0	4336
180	99.3	75.9	100.0	93.0	20.0	99.5	94.5	69.6	99.5	95.0	0.0	4860
200	99.2	77.8	100.0	91.0	30.0	100.0	93.5	73.9	99.5	96.0	2.2	5384

the TPR and detection latency. To understand and analyze these tradeoffs, we determined the per-path TPR, average FPR, and detection latency for different window size configurations, ranging from 20 to 200, presented in Table III. As the window size increases from 20 to 80, the TPR decreases for all paths and all malware. With further increases in window size, the TPR for some paths increases. For example, the TPR for Path 3 for the File Manipulation malware increase from 99.0% to 99.5%. Further increasing the window size to 200 results in only two cases in which the larger window size achieves a higher TPR than a window size of 20, specifically Path 3 for the File Manipulation and Fuzz 20% malware. Additionally, with a window size of 200, the FPR increases from 0% to 2.2%. Finally, as the window size increases, the malware detection latency increases linearly with the window size, requiring approximately 26 cycles per sample. Even though we selected the window size to minimize FPR and hardware area, a window size of 20 on average yields the highest TPR for the malware considered.

VII. CONCLUSIONS AND FUTURE WORK

The CDF-based timing anomaly detection detects small deviations in a system's timing distribution with high accuracy, achieving an average detection rate of 96.15% for the malware considered. By determining a per operation and per path threshold, the presented approach yields a 0% false positive rate. The CDF-based anomaly detection further provides an estimate of the probability of malware that quantifies the possibility of deviation caused by malware execution, and potentially enables runtime mitigation methods to utilize this estimate to determine the appropriate actions. Future work will develop automated design methods to optimize the system's configuration given latency, energy, and area constraints.

VIII. REFERENCES

- [1] Evans, D. The Internet of Things: How the Next Evolution of the Internet Is Changing Everything. Cisco White Paper, 2013.
- [2] McAfee Labs. Threats Report: December, 2016.
- [3] Li, C., A. Raghunathan, N. K. Jha. Hijacking an insulin pump: Security attacks and defenses for a diabetes therapy system. Conference on e-Health Networking Applications and Services, pp. 150-156, 2011.
- [4] Holm H. Signature Based Intrusion Detection for Zero-Day Attacks: (Not) A Closed Chapter? Hawaii International Conf. on System Sciences, 2014.
- [5] Chandola, V., A. Banerjee, V. Kumar. Anomaly Detection: A Survey. ACM Computing Survey, 41(3), 2009
- [6] Zhang, T., X. Zhuang, S. Pande, W. Lee. Anomalous Path Detection with Hardware Support. Conference on Compilers, Architectures and Synthesis for Embedded Systems, pp. 43-54, 2005.
- [7] Arora, D. A. Raghunathan, S. Ravi, N. K. Jha. Architectural Support for Safe Software Execution on Embedded Processors. Conference on Hardware Software Co-design and System Synthesis, pp. 106- 111, 2006.
- [8] Wagner, D., P. Soto. Mimicry Attacks on Host based Intrusion Detection Systems. Conf. on Computer and Communications Security, pp. 255-264, 2002.
- [9] Kruegel, C. E. Kirda, D. Mutz, W. Robertson, G. Vigna. Automating Mimicry Attacks using Static Binary Analysis. USENIX Security Symposium, pp.161-176, 2005.
- [10] Zimmer, C., B. Bhat, F. Mueller, S. Mohan. Time-Based Intrusion Detection in Cyber-Physical Systems. ACM/IEEE International Conference on Cyber-Physical Systems (ICCPs), pp. 109-118, 2010.
- [11] Yoon M.-K., S. Mohan, J. Choi, J.-E. Kim, L. Sha. SecureCore: A Multi-core-based Intrusion Detection Architecture for Real-Time Embedded Systems. Real-Time and Embedded Technology and Applications Symposium, 2013.
- [12] Lu, S., R. Lysecky. Time and Sequence Integrated Runtime Anomaly Detection for Embedded Systems. ACM Transactions on Embedded Computing Systems, 17(2), Article 38, pp. 1-27, 2017.
- [13] Patel K., S. Parameswaran, SHIELD: A Software Hardware Design Methodology for Security and Reliability of MPSOCs. Design Automation Conference, pp. 858-861, 2008.
- [14] Patel, K., S. Parameswaran, R. Ragel. Architectural Frameworks for Security and Reliability of MPSOCs. IEEE Transactions on Very Large Scale Integration Systems, No. 99, pp. 1-14, 2010.
- [15] Sametinger J., J. Rozenblit, R. Lysecky, P. Ott. Security Challenges for Medical Devices. Communication of ACM, 58(4), pp.74-82, 2015.
- [16] Wasicek A., P. Derler, E. A. Lee. Aspect-oriented Modeling of Attacks in Automotive Cyber-Physical Systems, Design Automation Conference, pp. 1-6, 2014.
- [17] Stollon, N. On-Chip Instrumentation: Design and Debug for Systems on Chip. Springer US, 2011.
- [18] Jiang, Z., M. Pajic, S. Moarref, R. Alur, R. Mangharam. Modeling and Verification of a Dual Chamber Implantable Pacemaker. Conf. on Tools and Algorithms for the Construction and Analysis of Systems, pp. 188-203, 2012.
- [19] Singh, N.K., A. Wellings, A. Cavalcanti. The Cardiac Pacemaker Case Study and its Implementation in Safety-Critical Java and Ravenscar Ada. Workshop on Java Technologies for Real-time and Embedded Systems, 2012.
- [20] Xilinx, Inc. MicroBlaze Processor Reference Guide, UG984. 2016.
- [21] Wilhelm, R., J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaat, P. Puschner, J. Staschulat, P. Stenstrom. The Worst-Case Execution-Time Problem - Overview of Methods and Survey of Tools. ACM Transactions on Embedded Computing Systems, 7(36), pp. 1-47, 2008.
- [22] Chakravarti, Laha, and Roy, (1967). Handbook of Methods of Applied Statistics, Volume I, John Wiley and Sons, pp. 392-394.