Statistical Time-based Intrusion Detection in Embedded Systems

Nadir A. Carreón, Allison Gilbreath, Roman Lysecky
Department of Electrical and Computer Engineering, University of Arizona, Tucson, Arizona, USA
nadir@email.arizona.edu, alligilbreath@email.arizona.edu, rlysecky@ece.arizona.edu

This paper presents a statistical method based on cumulative distribution functions (CDF) to analyze an embedded system's behavior to detect anomalous and malicious executions behaviors. The proposed method analyzes the internal timing of the system by monitoring individual operations and sequences of operations, wherein the timing of operations is decomposed into multiple timing subcomponents. Creating the normal model of the system utilizing the internal timing adds resilience to zero-day attacks, and mimicry malware. The combination of CDF-based statistical analysis and timing subcomponents enable both higher detection rates and lower false positives rates. We demonstrate the effectiveness of the approach and compare to several state-of-the-art malware detection methods using two embedded systems benchmarks, namely a network connected pacemaker and an unmanned aerial vehicle, utilizing seven different malware.

Keywords—Embedded systems security, anomaly-based detection, runtime intrusion detection, timing-based detection

I. Introduction

Internet connected devices have grown explosively and are estimated to reach several billions by 2020 [1]. Simultaneously, more than 140 million new malware were created in 2018 [2], which is indicative of the scope of the potential threat to security and privacy. Connected embedded systems can no longer rely on physical isolation to ensure security. Many of these systems are used in healthcare, where internet connectivity enable users and physicians to remotely access devices. Malicious attacks can have significant impacts both to patients' privacy and safety. Numerous vulnerabilities have been demonstrated in medical devices, and other life-critical systems. 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. McCarthy et al. [4] demonstrated that disabling an automobile's brake system was possible.

Broadly defined, there are two types of essential security measures for securing embedded systems. Proactive measures commonly include communication protocols and static application security testing, among others, but system designers must also consider scenarios in which attackers can circumvent a system's defenses. Reactive security measures are also needed to analyze the system at runtime to determine if malware is affecting the system execution. Signature-based detection systems (e.g. an antivirus running in your computer) focus on matching known traces in its database, because of this it is not able to protect against newly found threats (i.e. zero-day attacks [5]). Anomaly-based detection systems create a model based on the normal behavior of the system and detect any deviation from the expected behavior. One of the most common approaches is monitoring the execution sequence of the system's function calls [6][7][8]. Every time a function is executed out of the expected order, sequence-based anomaly detection flags it as malware. However, these type of anomaly detectors do not provide protection against mimicry malware [9][10]. Mimicry avoids detection by interleaving malicious functions within normal functions, avoiding executing functions out of order, but successfully carrying out the attack by eventually reaching the necessary functions following the normal execution sequence.

Timing-based anomaly detection [18][19] exploits the fact that many embedded systems have tight timing constraints to additionally detect deviations in the execution times of a system's operations. Although mimicry malware doesn't execute any function out of the expected order, the malicious actions have a noticeable effect on the internal timing of the system. Thus, utilizing a system model that combine sequence and timing based execution enables more robust malware detection against mimicry malware.

However, existing timing-based anomaly detection primarily relies on so called lumped timing measurements [19] that can have significant variations due to the timing overhead incurred by interrupts, cache misses, and the system architecture. Alternatively, the timing data can be broken down into subcomponents with tighter distributions that enable better detection. A timing model with three subcomponents splits the timing measurement into different measurable subcomponents, including the intrinsic software execution (i.e. ideal software execution time without the overhead due to the OS, interrupts, etc.), the time overhead due to the instruction cache (I$) misses, and the timing overhead due to data cache (D$) misses.

This paper presents a statistical timing-based anomaly detection method that analyzes the cumulative distribution functions (CDF) [16] of timing subcomponents within sliding execution windows. As the approach detects deviations in execution timing, it can also indirectly detect other malware types (i.e. non-mimicry malware). We further present a path-based method for estimating the probability of malware based on observed timing measurements at runtime, and we present a methodology for defining probabilities thresholds to minimize false positive rates, which is a critical concern for any anomaly-based detection method. The novelty resides on analyzing the subcomponent timing using cumulative distribution functions along execution sequences, to create a model of the system.

II. Related Work

Several efforts have been made to develop sequence-based anomaly detectors. Patel et al. [21][22] use a control flow map to monitor execution sequences of the target system. However, their detection methods require code instrumentation to transmit

This research was partially supported by the National Science Foundation under Grant CNS-1615890.

measurements to a dedicated processor for analysis, which incurs up to a 44% performance overhead and 27% area overhead. Embedded systems usually have very tight constraints, especially medical systems. Thus, these overheads may be infeasible or cost prohibitive.

Although sequence-based detectors can achieve good detection rates, they are unable to protect the system against mimicry attacks. Wagner et al. [9][10] demonstrated that sequence-based detectors are unable to detect mimicry malware. They did this by evaluating different sequence-based detectors against mimicry malware. However, although mimicry malware doesn't alter the execution sequence, it still influences the internal timing. The mimicked normal operations commonly use dummy or null parameters, and the malicious operations change the internal functionality of said operations, which impacts the time the operation takes to execute. Even small changes in the timing of execution of a single function can have measurable impacts to the timing of other operations.

Creating a model of the system by analyzing the timing behaviors provides additional resilience against mimicry malware, as has been demonstrated by several existing methods.

Lu et al. [19][20] defined several methods to analyze the timing distribution and create anomaly detectors using non-intrusive hardware. Their approaches include a distance-based approach and a support vector machine approach. The granularity of their approach is at a subcomponent-level, in which they separate the timing information into three different times, namely D$, I$, and intrinsic timing. However, their approach separately analyzes the timing behavior of each execution of each operation, meaning that it does not consider the statistical distribution of timing behaviors nor the timing of entire execution paths.

Yoon et al. [18] presented SecureCore, which focuses on analyzing the timing distribution at the basic block level. At runtime, the timing of the basic block is measured, and the probability of that measurement being observed is estimated. If the probability is below a previously defined threshold, the execution is flagged as malicious. Since false positives may deteriorate the performance of the system, defining the correct thresholds is critical. However, it must be noted that false positives are inherent in this approach, and when applied to operations at a coarse granularity (e.g. system or function calls) the false positive rate can be as high as 10%.

Other efforts include monitoring the control flow sequences within sliding windows [7], analyzing the data dependencies between internal operations by monitoring the dataflow of an application [23], creating stochastic models based on Markov chains [24] that describe any possible sequence of events along with their probabilities, and modelling the dataflow as the relationship between the arguments for different function calls or control flow operations [25][26]. However, these would be infeasible in our target systems due to performance or area overheads, data not being accessible at runtime, or their susceptibility to mimicry malware.

III. Assumptions and Threat Model

To appropriately evaluate the proposed model, several assumptions about the target systems were made. We assume the considered malware has already been able to gain access to the system. How the malware was able to gain access to the system is out of the scope for this paper, and it doesn't influence the evaluation of our proposed model. All malware were implemented as mimicry malware, done by interleaving malicious operations within normal operations. Since mimicry malware does not violate the execution sequence, therefore, a sequence-based anomaly detector would not be able to detect any of these malware.

Seven mimicry malware [11][12][13][14][15] were implemented targeting prototypes of a network connected pacemaker and an unmanned aerial vehicle (UAV). These malware all represent real threats from different systems that have been modified to attack our systems. The *Fuzz Malware* interferes with the normal behavior by "fuzzing" (i.e. randomizing) the data within the pacemaker's cardiac log. This malware is implemented at two different fuzzification levels, namely 20% and 100%. The *Information Leakage* malware breaks confidentiality by sending private patient's data from the pacemaker to an unauthorized party. A *Data Manipulation* malware modifies data stored inside both target systems. For the pacemaker, this malware modifies the cardiac log so as to mislead a physician. For the UAV, the malware manipulates the encrypted files that causes a failure when decrypting them. The *Gain Scheduling* malware manipulates the gains or gain scheduling logic, which could cause decreased performance or dangerous instability in the control systems of the UAV. The *Image Fuzz* malware renders an image unusable by adding a Gaussian noise during the compression process onboard the UAV. The *Camera Obfuscation* malware interferes with the correct functionality of the UAV's camera by altering the behavior of the camera's flash (i.e. increasing or disabling the flash), rendering the image unusable. Finally, the *Key Leakage* malware breaks confidentiality by stealing and transmitting the encryption keys of the UAV to an unauthorized party.

Fig. 1 shows an overview of the proposed system architecture and methodology for CDF-based anomaly detection

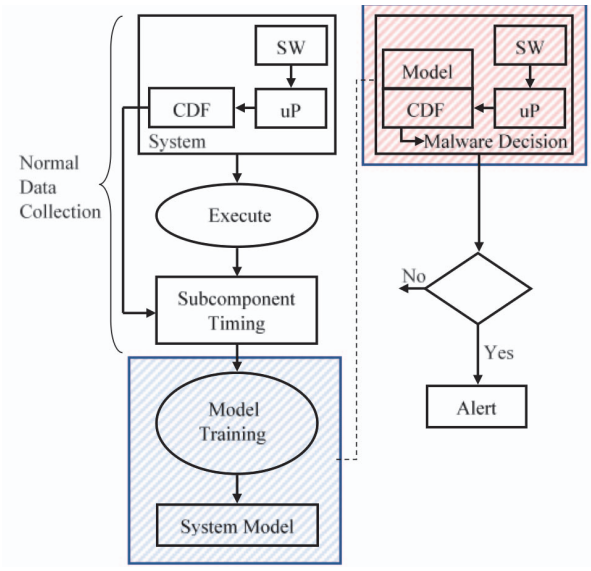

Fig. 1. CDF-based anomaly detection using timing subcomponents.

using timing subcomponents. The system is first executed under normal circumstances to collect the required amount of timing samples to create a model of the expected system execution behavior. A CDF hardware component analyzes the system non-intrusively by directly interfacing with the microprocessor's trace port, measures the time each operation takes to execute, and extracts the timing data of the different subcomponents.

Once the required amount of timing samples has been collected, the CDF-Analysis (highlighted in blue) creates the normal model of the system. The model is created by analyzing the timing distribution of the different subcomponents using Cumulative Distribution Functions (CDFs). This model is stored inside the CDF component to be used at runtime to determine if the runtime execution matches the expected CDF model or if timing deviation are due to malicious activities.

At runtime (highlighted in red) the state of the system is analyzed by looking at the state of the microprocessor through the trace port. Once enough timing samples have been collected, the CDF-Analysis is executed and the timing distribution is compared against the stored model. If the state of the system deviates beyond a previously defined threshold, an alert is sent.

IV. Timing and Classification Models

The system is monitored by analyzing the state of the microprocessor by directly interfacing with it through its trace port. This approach is non-intrusive in the sense that the observation method does not affect the system behavior (i.e. no performance overhead is incurred). The CDF component detect the operation's execution by observing the program counter (PC) and looking for matching addresses against previously known addresses. Whenever an address matches the start address of an operation a timer starts, and whenever an address matches an end address, said timer stops.

Time measurements can be done at different granularities (e.g. operation-level, subcomponent-level). Building the model utilizing a finer granularity yield tighter bounds, which in turn improves accuracy. We generically define an operation as a function call of the system. We define the intrinsic timing as the ideal execution time of the software without any overhead produced by the environment (i.e. OS, cache misses). On the other hand, the incidental timing is the overhead produced by the environment in which the software executes. The incidental timing is further broken down into the timing overhead produced by the I$ and D$ misses. Our approach measures the time of three different subcomponents: intrinsic, D$, and I$ timing.

Using these timing measurements, several classification models can be used to determine if the execution is malicious. We present on overview of two existing subcomponent timing classification from [20], including a Euclidean distance-based model and support vector machine (SVM) model, which we compared to in our experimental results. We then present our proposed new methods using CDF-based statistical analysis of execution windows, one which performs malware classification at the operation level and one at the execution path level.

Distance based subcomponent (DBS) evaluates the data from each subcomponent together. The data points are arranged in a multi-dimensional feature space, with each dimension representing the timing data of one subcomponent (i.e. intrinsic, I$, D$). Clusters (i.e. spheres) are created from the data points utilizing the Euclidian distance, with the center of each one being their mean, and the farthest data point from the mean determining its radius. In this approach, every value that falls outside a sphere's radius is flagged as malware. Hierarchical clustering is used to cluster the normal data into multiple spheres, where the number of spheres has a direct effect on the tradeoff between detection accuracy and false positives. For this analysis, the number of spheres was set to 8, which yields a false positive rate below 1% for both target systems.

Support Vector Machines (SVM) use supervised learning theory for binary classification, which have been extensively used in the field of bioinformatics and image recognition. For anomaly detection, where only a model of normal system execution exists, the decision of whether the incoming data belongs to the one normal class can be solved using a one-class SVM [6], which are particularly useful when a higher number of features to train the system are available. In this case, the timing subcomponents are the features used by the SVM. The advantage of SVM against other existing machine learning techniques is its lower complexity [27] and requirement of only needing normal timing measurements to train the model. Using the Schölkopf's model [28] all features of the training data (i.e. subcomponent timing data) are mapped into a hyperplane, maximizing the distance between the hyperplane and the original feature space. A binary function is then used to make the decision whether malware is executing inside the system.

We present a new *Operation-based Historical CDF (OHCDF)* based malware classification model that makes malware classification at the operation level. The CDF-Analysis is based on the Kolmogorov-Smirnov test (K-S Test) [17], both of which focus on analyzing the data distribution. However, a traditional K-S test requires collecting thousands of timing samples before testing distributions, which is infeasible due to the resources and time needed to collect and store the data. Instead, we use a modified K-S test to statistically analyze the

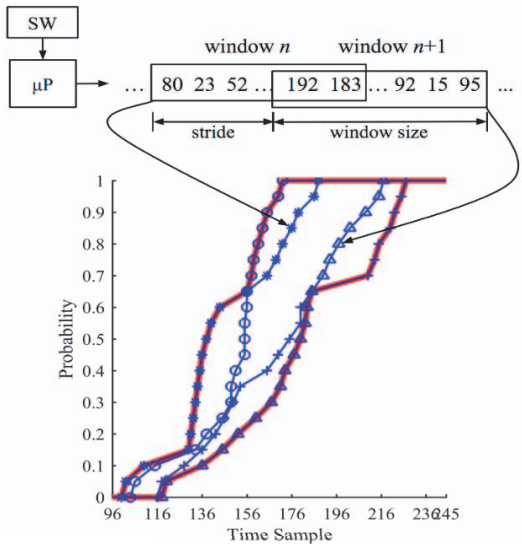

Fig. 2. Conceptual overview of window-based CDF calculation, highlighting the sliding windows of timing samples used to calculate the CDF of an operation. The plot shows the CDFs for four different windows, from which the upper and lower CDF boundaries can be calculated.

CDFs of operation's timing within a fixed size sliding execution window. Each window has two properties, the window size defines how many samples are stored inside the window at any given time, and the stride defines how many new samples needs to be recollected before executing the CDF analysis again.

The CDF-Analysis consist of two parts, constructing the model of the system, and calculating the runtime threshold. The normal data collected to train the system is split into two sets, the first is used to create the model of the system, and the second to calculate the threshold of the subcomponents per operation.

1) Boundary Construction

The timing samples of the first data set are split into small segments (i.e. windows). Each window overlaps 75% with the previous one, and having 25% of new timing samples (e.g. a stride of 5 for a window size of 20). The CDF of each window is then computed and compared against each other. The most extreme points found across all their CDFs, as shown in Fig. 2, define the CDF bounds of the normal behavior for an operation. These boundaries are used at runtime to estimate the probability of the execution being malicious based on how much it deviates from the CDF boundaries. The estimated probability of malware is defined as the percentage of the runtime CDF that falls outside the CDF boundaries, calculated as the complement of the overlap between the runtime CDF and the CDF boundaries.

2) Threshold Calculation

The second set of normal timing measurements is also split into overlapping windows and their CDFs are computed. For each window of each subcomponent, the complement of the overlap between their CDF and the subcomponent's CDF boundaries is calculated. The obtained value reflects how different the data inside the window is from the expected behavior. Next, the system finds the window that deviates the most from the expected behavior and set that as the threshold. In our approach, the threshold defines the maximum deviation the runtime CDF can have, while still being considered normal.

At runtime, once enough data samples have been collected to fill a window, the CDF for each subcomponent is computed and compared against the subcomponent's previously defined CDF boundaries. The estimated probability of malware ($P_{est}o_i(M)$) is then calculated and compared against the subcomponent's threshold. An example of three different execution windows can be seen in Fig. 3. If the estimated probability of malware is above the threshold the execution

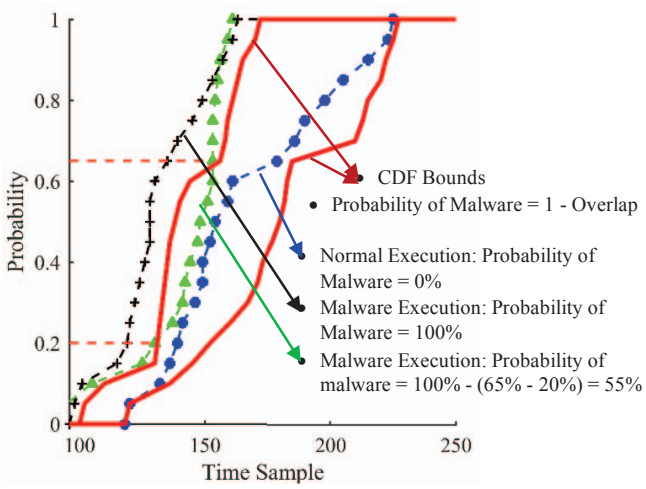

Fig. 3. Three runtime CDFs plotted with the upper and lower CDF bounds (red solid lines), showing the overlap used to calculate the estimated probability of malware. Normal execution of the system (blue circles) is always inside the previously obtained boundaries. The CDFs for malware (black crosses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

window is flagged as malware, and a non-maskable interrupt is sent. Analyzing the timing data at the subcomponent level may reveal malware that would not be detected by a lumped timing as can be seen on Fig. 4. Profiling the system using the subcomponent timing data yields a finer granularity, which increases the accuracy of the anomaly detector. At runtime, if any of the subcomponents of an operation flags the execution window as malware, the operation is said to be malicious.

The OHCDF approach performs classification at the operation level, which can achieve high detection rates but with higher than desired false positive rates. Thus, we further extend this approach to create *Path-based Historical CDF (PHCDF)* that perform malware classification by aggregating the estimated probabilities of malware of several operations inside a path, yielding a more robust decision. A path is defined as a specific sequence of operations that execute at the same rate inside a software task. The path-based CDF analysis has an extra step compared to the OHCDF analysis, which is the aggregation of the thresholds and runtime CDFs per path.

3) Path Malware Probability Calculation

Estimating the probability of malware of an execution path ($P_{est}p_j(M)$) depends on the independent probabilities of malware

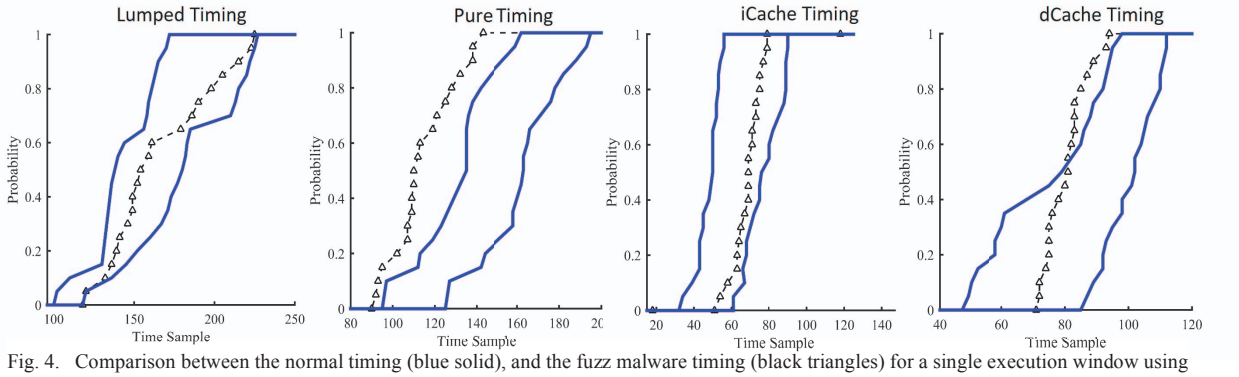

Fig. 4. Comparison between the normal timing (blue solid), and the fuzz malware timing (black triangles) for a single execution window using the lumped timing and the subcomponent timing models. The malware is detected 100% of the time by the intrinsic timing, while is would not be detected by the lumped timing.

TABLE 1: NUMBER OF TASKS/ISRS, NUMBER OF OPERATIONS, AND MALWARE AFFECTING THE PACEMAKER AND UAV TARGET SYSTEMS.

	Pacemaker	*UAV*
# Operations	43	51
# Tasks/ISRs	3/4	5/0
Affecting Malware	Fuzz 20%, Fuzz 100%, File Manipulation, Information Leakage	Gain Scheduling, Image Fuzz, Data Manipulation, Key Leakage, Camera Obfuscation

of the operations inside the path. The estimated probability of malware of a path is calculated using (1), defined as the probability that at least one of the operations is malicious.

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

4) Path Threshold Calculation

Previously, we had shown that the decision whether the execution was normal or malicious was made by comparing the estimated probability of malware against the threshold. Similarly, the decision whether the path execution is malicious or not is made by comparing the path's estimated probability of malware against the path's threshold (Tp_j). Calculating the path threshold is similar to calculating the estimated probability of malware of the path. However, instead of utilizing the estimated probabilities of malware, we utilize the previously calculated thresholds ($P_{max}o_i(M)$) of all operations inside the path, as shown in (2). The resulting threshold is rigorous enough to accurately distinguish normal from malicious executions.

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

At runtime, once enough timing data has been collected to fill the windows of each operation in the path, the subcomponent's CDFs are calculated and compared against the previously defined CDF boundaries to determine their individual probabilities of malware. The estimated probability of malware of the execution path is then calculated and compared against the t path threshold. If the estimated probability of malware is above the threshold the execution path is flagged as malware, and a non-maskable interrupt is sent. Like in the previous approach, this process is repeated for the three subcomponents considered, with any subcomponent flagging the execution as malicious resulting in the path determined to be malicious.

V. EXPERIMENTAL RESULTS

To demonstrate the advantages our CDF-based analysis of timing subcomponents compared to existing methods, we conducted experiments for two target systems with a total of seven mimicry malware using a window size of 20 and a stride of 5. Table 1 shows both target systems considered along with a summary of their characteristics. The *Unmanned Aerial Vehicle (UAV)* [29] is a prototype for an autonomous drone capable of automatically (or under manual control) collect images for target locations. The *Pacemaker* [20] is a prototype for a connected pacemaker featuring remote connections for both automated communication of cardiac activity to healthcare providers and configurable pacing parameters by physicians. Across all malware classification methods considered, any operation with an individual false positive rate above 5% was excluded from the timing model (i.e the operation was not monitored at runtime). We cross validated (k = 10) to determine the detection and false positive rates using 1000 executions per malware for the corresponding target system.

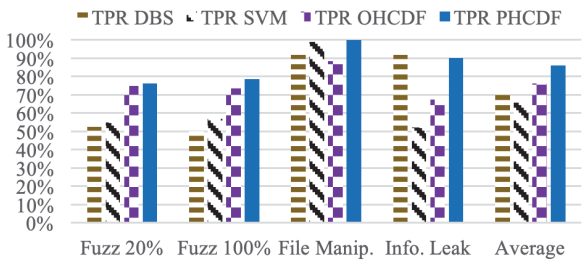

Fig. 5. Detection rates for all malware affecting the pacemaker.

We designed a hardware implementation of PHCDF malware detector and synthetized the designed targeting an Artix-7 XC7A200T FPGA with Vivado 2016.4. The hardware supports 51 operations, using 32-bit registers for both the operation's addresses and timers, and using block RAMs (BRAMs) as local memories. The hardware requires 7,979 lookup tables (LUTs), 9,100 flip-flops (FF), and three BRAMs (1x254Kb, 2x512Kb). No additional external memory or storage is needed. The hardware component has a maximum operating frequency of 111 MHz, which is sufficient for the target integration with a 100 MHz MicroBlaze based system. The hardware component has a peak power consumption of 112 mW (with 65mW being consumed by the BRAMs), and average power consumption of 80.74 mW, which corresponds to a power overhead of only 6.89%. We note that only the event detection and timing components need to operate at the processor frequency and using a dual clock configuration could enable lower energy consumption, although that is left as future work.

Fig. 5 presents the detection rates for each one of the malware affecting the pacemaker, for the four different approaches previously discussed. The SVM has the lowest average detection rate across all malware with 65.8%, although it's important to note that this is due to SVM having a very low detection rate for the Information Leakage malware, achieving just 52.0%. On the other hand, PHCDF achieves the highest average detection rate of 86.2%, with the highest detection rate for all malware except the Information Leakage malware. PHCDF achieves a detection rate of 76.22%, 78.57%, and 100% for Fuzz 20%, Fuzz 100%, and File Manipulation malware, respectively. While PHCDF achieves a 90% detection rate for the Information Leakage malware, DBS achieves a higher detection rate of 92.0%. In contrast, OHCDF has the second highest detection rate with an average of 76.1%, DBS achieves the third highest with 71.6%, and SVM the lowest with 65.8%.

For the UAV, as shown in Fig. 6, OHCDF achieve the lowest average detection rate with 76.0%. PHCDF achieved again the highest average detection rate with 89.3%, having the highest detection rates on all analyzed malware except Gain Scheduling. However, it must be noted that Gain Scheduling might be an outlier, since the operation selection criteria left out most of the affected operations, lowering the detection rate drastically by up to 62.7%, as opposed to 83.8% for Image Fuzz, and 100% detection rates for the Data Manipulation, Key Leakage, and Camera Obfuscation malware. SVM yields the second highest detection rate with 86.8%, followed by DBS with 79.2%.

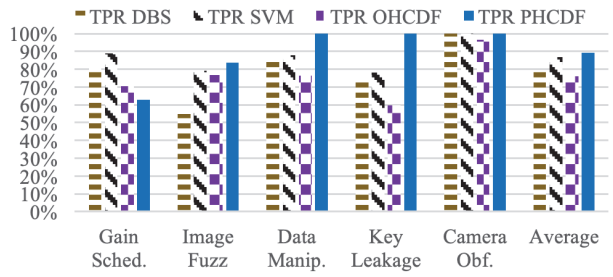

Fig. 6. Detection rates for all malware affecting the UAV.

TABLE 2: AVERAGE FALSE POSITIVE RATE FOR ALL DIFFERENT APPROACHES, FOR BOTH THE PACEMAKER AND UAV TARGET SYSTEMS.

	Pacemaker	*UAV*
DBS	0.0022	0.0079
SVM	0.0107	0.0089
OHCDF	0.0325	0.0245
PHCDF	0.0007	0.0017

Table 2 shows the average false positive rate for both systems monitoring only operations whose individual false positive rate is below 5%. OHCDF yield false positive rates of up to 3.25% for the pacemaker, and 2.45% for the UAV, which is 2.18% and 1.56% higher than the second highest false positive rate for the pacemaker and UAV, respectively. Although SVM yields high detection rates, its false positive rates are the second highest for both target systems, yielding 1.07% and 0.89% for the pacemaker and UAV respectively. On the other hand, PHCDF yields the lowest false positive rates for both target systems, with 0.07% for the pacemaker, and 0.17% for the UAV.

VI. CONCLUSIONS

Path-based statistical analysis of subcomponent timing yields higher detection rates compared both to statistical analysis of individual operations and previously presented methods using Euclidean distance and support vector machines. Aggregating the results from the PHCDF analysis across all subcomponents yield high detection rates along with very low detection rates at the cost of a small increase in latency due to the system needing to gather enough new data samples before executing the CDF analysis. For the considered target systems, the PHCDF malware detection achieves the highest average detection rates with the lowest average false positive rates. Specifically, PHCDF achieves 14.6% and 2.5% higher detection rates and 0.15% and 0.72% lower false positive rates compared to DBS and SVM for the pacemaker and UAV target systems, respectively.

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, 2018.

[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] C. McCarthy, K. Harnett, A. Carter. Characterization of Potential Security Threats in Modern Automobiles: A Composite Modeling Approach. National Highway Traffic Safety Administration, Washington Tech. Rep, (Oct. 2014).

[5] Holm H. Signature Based Intrusion Detection for Zero-Day Attacks: (Not) A Closed Chapter? Hawaii International Conf. on System Sciences, 2014.

[6] Chandola, V., A. Banerjee, V. Kumar. Anomaly Detection: A Survey. ACM Computing Survey, 41(3), 2009

[7] 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.

[8] 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.

[9] Wagner, D., P. Soto. Mimicry Attacks on Host based Intrusion Detection Systems. Conf. on Computer and Comm. Security, pp. 255-264, 2002.

[10] Kruegel, C. E. Kirda, D. Mutz, W. Robertson, G. Vigna. Automating Mimicry Attacks using Static Binary Analysis. USENIX Security Symposium, pp.161-176, 2005.

[11] Hartmann, K., and C. Steup, The vulnerability of UAVs to cyber attacks—An approach to the risk assessment. Conf. on Cyber Conflict, 2013.

[12] Kim A., B. J. Wampler I., Goppert, Hwang, and H. Aldridge, Cyber attack vulnerabilities analysis for unmanned aerial vehicles. The American Institute of Aeronautics and Astronautics: Reston, VA, USA, 2012.

[13] Sun S., S. Kwong, B. Lei, S. Zheng. Watermarking Based on Stochastic Resonance Signal Processor. Pacific Rim Cond. on Multimedia, pp: 367-375. 2007.

[14] Sametinger J., J. Rozenblit, R. Lysecky, P. Ott. Security Challenges for Medical Devices. Communication of ACM (CACM), Vol. 58 No. 4, pp. 74-82, 2015.

[15] Wasicek A., P. Derler, E. A. Lee, Aspect-oriented Modeling of Attacks in Automotive Cyber-Physical Systems, Design Automation Conference, pp. 1-6, 2014.

[16] K. L. Monti. Folded Empirical Distribution Function Curves-Mountain Plots, The American Statistician Vol. 49, No. 4, pp. 342-345, 1995.

[17] Chakravarti, Laha, and Roy, (1967). Handbook of Methods of Applied Statistics, Volume I, John Wiley and Sons, pp. 392-394.

[18] Yoon M.-K., S. Mohan, J. Choi, J.-E. Kim, L. Sha. SecureCore: A Multicore-based Intrusion Detection Architecture for Real-Time Embedded Systems. Real-Time and Embedded Technology and Applications Symposium, 2013.

[19] Lu, S., Lysecky R., Rozenblit J. Subcomponent Timing-Based Detection of Malware in Embedded Systems. IEEE International Conference on Computer Design (ICCD), pp. 17-24, 2017.

[20] Lu, S., R. Lysecky. Data-driven Anomaly Detection with Timing Features for Embedded Systems. ACM Trans. Des. Autom. Electron. Syst. 24, 3, Article 33 (April 2019), 27 pages.

[21] Patel K., S. Parameswaran. SHIELD: A Software Hardware Design Methodology for Security and Reliability of MPSOCs. Design Automation Conference, pp. 858–861, 2008.

[22] 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.

[23] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. Iyer. Non-control-data attacks are realistic threats. USENIX Security Symp., pp. 177–192, 2005.

[24] A. Frossi, F. Maggi, G. Rizzo, and S. Zanero. Selecting and improving system call models for anomaly detection. Conf. on Detection of Intrusions and Malware, and Vulnerability, pp. 206–223, 2009.

[25] M. Bond, V. K. Srivastava, K. McKinley, and V. Shmatikov. Efficient, context-sensitive detection of real-world semantic attacks. Programming Languages and Analysis for Security, pp. 1–10, 2010..

[26] S. Bhatkar, A. Chaturvedi, and R. Sekar. Dataflow anomaly detection. In Symposium on Security and Privacy, pp. 15–62, 2006.

[27] Kulkarni A., Y. Pino, M. French, T. Mohsenin, Real-time anomaly detection framework for many-core router through machine learning techniques. Journal on Emerging Technologies in Computing Systems, 2016.

[28] Schlkopf B., R.C. Williamson, A.J. Smola, J. Shawe-Taylor, J. Platt. Support Vector Method for Novelty Detection, Advances in Neural Information Processing Systems 12, pp. 526-532, 1999.

[29] Frost and Sullivan. Study Analysing The Current Activities in The Field of UAV. Tech. Rep., European Commission Enterprise and Industry Directorate-General, 2007.