Learning Execution Contexts from System Call Distribution for Anomaly Detection in Smart Embedded System

Man-Ki Yoon∗, Sibin Mohan†, Jaesik Choi‡, Mihai Christodorescu§ and Lui Sha∗
∗Dept. of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL
†Information Trust Institute, University of Illinois at Urbana-Champaign, Urbana, IL
‡School of Electrical and Computer Engineering, Ulsan National Institute of Science and Technology, Ulsan, Korea
§Qualcomm Research Silicon Valley, Santa Clara, CA
Email: {mkyoon, †sibin, §lrs}@illinois.edu, ‡jaesik@unist.ac.kr, §mihai@qti.qualcomm.com

Abstract—Existing techniques used for anomaly detection do not fully utilize the intrinsic properties of embedded devices. In this paper, we propose a lightweight method for detecting anomalous executions using a distribution of system call frequencies. We use a cluster analysis to learn the legitimate execution contexts of embedded applications and then monitor them at run-time to capture abnormal executions. Our prototype applied to a real-world open-source embedded application shows that the proposed method can effectively detect anomalous executions without relying on sophisticated analyses or affecting the critical execution paths.

I. INTRODUCTION

With advanced functionality and connectivity enabled by modern computing and communication technologies, embedded devices are ubiquitously networked as an important component for Internet of Things (IoT). Open-source software (both operating system and applications) plays an important role in the IoT software ecosystem, enabling community-supported development of smart embedded applications. However, this open nature of the software environments, in conjunction with the increased capabilities and complexities of the modern embedded systems, introduce more security threats. As demonstrated by recent attacks [1]–[4], threats to these systems are growing, both in number as well as sophistication.

The increasing security challenges posed on these systems make it virtually impossible to completely secure them due to many entry points that are vulnerable to potential security threats. Thus, instead of attempting to prevent every possible security breach, we intend to detect anomalies by monitoring the behavior of the application; deviation from expected behavior is considered malicious [5]. Traditional behavior-based anomaly detection systems rely on specific signals such as network traffic [6], [7], control flow [8], [9], system calls [10]–[12], etc. The use of system calls, especially in the form of sequences [10], [13]–[17], has been extensively studied in behavior-based anomaly detection for general purpose systems since malicious activities often use system calls to execute privileged operations on system resources. Because server, desktop and mobile applications exhibit rich, wildly varying behaviors across executions, such methods need to rely either (a) on complex models of normal behavior, which are expensive to run and thus unsuitable for an embedded system, or (b) on simple, partial models, which validate only small windows of the application execution at a time. This opens the door for attacks where variations of a valid execution sequence are replayed with slightly different parameters to achieve a malicious goal; on the other hand, the application would not execute that sequence of operations in a normal manner, every time.

We observe that the very properties of embedded systems also make them amenable to the use of certain security mechanisms. The regularity in their execution patterns means that we can detect anomalies by monitoring the behavior of such applications [18]–[22] since the set of what constitutes legitimate behavior is often limited by design. In this paper we present an anomaly detection mechanism for embedded systems using a system call frequency distribution (SCFD). Figure 1 presents an example. It represents the numbers of occurrences of each system call type for each execution run of an application. The key idea is that the normal executions of an application whose behavior is regular can be modeled by a small set of distinct system call distributions (e.g., Figure 11 in Section V), each of which corresponds to a high-level execution context. We use a cluster analysis to learn distinct execution contexts from a set of SCFDs and to detect anomalous behavior.

Our detection method is lightweight, has a deterministic time complexity – hence, it fits well for resource-constrained embedded systems. This is due to the coarse-grained and concise representation of SCFDs. Although it can be used for offline analysis, we demonstrate an implementation on an embedded computing board [23] and show that minor modifications to the operating system and architectural supports from modern embedded processors enable us to monitor and analyze the run-time system call usage of applications in a secure, non-intrusive manner. We use a real-world open-source application [24] and demonstrate that SCFDs can effectively detect certain types of abnormal execution contexts that are difficult for traditional sequence-based approaches. Detailed results including the sequence-based security analysis is presented in Section V.

Hence, the high level contributions of this paper are:

1) we introduce a lightweight method, utilizing the predictable nature of embedded system behaviors, with a deterministic time complexity for detecting anomalous execution contexts of embedded systems based on the distribution of system call frequencies (SCFD). We use a cluster analysis to learn the legitimate execution contexts of embedded applications and then monitor them at run-time to capture abnormal executions. Our prototype applied to a real-world open-source application shows that the proposed method can effectively detect anomalous executions without relying on sophisticated analyses or affecting the critical execution paths.
2) we present a detailed security analysis on a real-world application and successful attacks that can fundamentally circumvent sequence-based detection methods;
3) we demonstrate our technique with the target application on an embedded computing board and evaluate its advantages and limitations using various attack scenarios.

II. OVERVIEW

The main idea behind SCFD is to learn the normal system call profiles, i.e., patterns in system call frequency distributions, collected during legitimate executions of a sanitized system. Analyzing profiles is challenging especially when such profiles change, often dramatically, depending on the execution modes, events, and inputs. We address this issue by clustering the distribution of system calls capturing legitimate behavior. Each cluster then can be a signature that represents a high-level execution context, either in a specific mode/event or for similar input data. Then, given an observation at run-time, we test how similar it is to each previously calculated cluster. If there is no strong statistical evidence that it is a result of a specific execution context then we consider the execution to be malicious with respect to the learned model.

Attacks against sequence-based approach: Although sequence-based methods can capture detailed, temporal relations in system call usages, they may fail to detect abnormal execution contexts. This is because sequence-based approaches fundamentally profile the local, temporal relations among system calls within a limited time frame. Figure 2 highlights such a case. The system calls shown in the figure are generated by Motion [24], an open-source motion detection application used in our evaluation. Each Motion loop saves the current motion frame to the filesystem if a motion is detected (the top block in the figure). A snapshot is saved too (the bottom block), independently, at a regular interval (e.g., once per 5 seconds). These two blocks use the same routine to save the images to files and thus generate same sequence of system calls as depicted. We were able to insert a small piece of code that leaks out the current motion frame to a desired location in the filesystem while making the resulting system call sequences still look legitimate (the detail is given in Section V-A). This was possible because (i) the sequence patterns generated only by the inserted block are identical to those made by the other two blocks (since the same routine is used) and (ii) no new patterns are generated by transitions across the blocks. Note that if only one of the legitimate blocks execute, the resulting sequences are still legitimate because the inserted block looks like the other block that did not execute. The only way a sequence-based approach can detect such a malicious execution is to know patterns that are long enough to learn the temporal relationship between the two legitimate blocks. That is, the expected sequence patterns must know what system calls should follow after two file operations. However, this is highly unlikely because the required pattern lengths are often too long and also can vary greatly due to variations in data (i.e., image) sizes.

An attacker who has access to the target application code can implement such a stealthy, malicious code that modifies the high-level execution context while not disturbing the system call sequence patterns. This is more probable especially when the target application has such a vulnerable structure as described above. In contrast to sequence-based techniques, our SCFD method can easily detect abnormal deviations in high-level, naturally variable execution contexts such as the one illustrated above (Figure 2) since the SCFD significantly changes due to the malicious execution; the SCFDs at the bottom in Figure 3 are clearly abnormal when compared to each of the normal patterns for Motion (the 11 patterns shown in the top of the figure). Also, if the attacker corrupts the integrity of the data (for instance, erases the motion frame so that no motion can be detected) then our method is able to detect it – this is not easy for sequence-based methods as we explain in Section V-A. Hence, by using these two approaches together, one can improve the overall accuracy of the system call-based anomaly detection.

Adversary Model: We consider threat models that involve changes to the behavior of system call usage. If an attack does not invoke or change any system calls, the activity at least has to not invoke or change any system calls, the activity at least has to

...
users to download the modified source code or the executable binary using, for example, a social engineering tactic. We do not focus on more active attacks such as process killing, privilege escalation, etc. as these will change the system call usage in an obvious way.

**Assumptions:** The following assumptions are made in this paper:

1) We consider applications that execute in a repetitive fashion which fits well for embedded applications (e.g., sensing and computation). Motion, used in our prototype and evaluation, is an example. We monitor and perform a legitimacy test at the end of each invocation of a task.

2) We limit ourselves to applications where most of the possible execution contexts can be profiled ahead of time. Hence, the behavior model is learned under the stationarity assumption – this is a general requirement of most behavior-based anomaly detection systems [5]. This can be justified by the fact that most embedded applications have a limited set of execution modes and input data falls within fairly narrow ranges. Also, a significant amount of analysis of embedded systems is carried out post-design/implementation anyways [26] for a variety of reasons. Hence, the information about the usage of system calls can be rolled into such a-priori analysis. Our method may not work well for applications that do not exhibit execution regularities but such systems are not the focus of our paper anyways.

3) The profiling is carried out prior to system deployment when the application is trustworthy. Also, any updates to the applications or the system must be accompanied by a repeat of the profiling process. We assume that the stored profile(s) cannot be tampered with (for example, by hardware-based protections [19], [27], [28]). As mentioned earlier, these assumptions must hold for any behavior-based monitoring/detection mechanisms.

**III. ANOMALY DETECTION USING EXECUTION CONTEXTS LEARNED FROM SYSTEM CALL DISTRIBUTIONS**

We now present our novel methods to detect abnormal execution contexts in embedded applications by monitoring changes in system call frequency distributions.

**A. Definitions**

Let \( \mathcal{S} = \{s_1, s_2, \ldots, s_D \} \) be the set of all system calls provided by an operating system, where \( s_d \) represents the system call of type \( d \). During the \( n^{th} \) execution of an application, it calls a multiset \( \sigma^n \) of \( \mathcal{S} \). Let us denote the \( n^{th} \) system call frequency distribution (or just system call distribution) as \( x^n = [m(\sigma^n, s_1), m(\sigma^n, s_2), \ldots, m(\sigma^n, s_D)]^T \), where \( m(\sigma^n, s_d) \) is the multiplicity of the system call of type \( d \) in \( \sigma^n \). Hereafter, we simplify \( m(\sigma^n, s_d) \) as \( x^n_d \). Thus, \( x^n = [x^n_1, x^n_2, \ldots, x^n_D]^T \).

We define a training set, i.e., the execution profiles of a sanitized system, as a set of \( N \) system call frequency distributions collected from \( N \) executions, and is denoted by \( \mathbf{X} = [x^1, x^2, \ldots, x^N]^T \). The clustering algorithm (Section III-C) then maps each \( x^n \in \mathbb{N}^D \) to a cluster \( c_l \in \mathcal{C} = \{c_1, c_2, \ldots, c_k \} \). We denote by \( c : \{x^1, \ldots, x^N\} \rightarrow \mathcal{C} \) the cluster that \( x^n \in \mathbf{X} \) belongs to.

![Fig. 4. System call frequency distributions for \( \mathcal{S} = \{s_1, s_2\} \) and clusters. The gray-colored objects are SCFDs in the training set. Each star-shaped point inside each cluster is its centroid. The ellipsoid around each cluster draws the cutoff line of the cluster; the points inside of the line are legitimate with respect to the cluster.](image)

**B. Learning a Single Execution Context**

The variations in the usage of system calls will be limited if the application under monitoring has a simple execution context. In such a case, it is reasonable to consider that the executions follow a certain distribution of system call frequencies, clustered around a centroid, and cause a small variation from it due to, for example, input data or execution flow. This is a valid model for many embedded systems since the code in such systems tends to be fairly limited in what it can do. Hence, such analysis is quite powerful in detecting variations and thus catching anomalies.

For a multivariate distribution, the mean vector \( \mu = [\mu_1, \mu_2, \ldots, \mu_D]^T \), where \( \mu_d = (\sum_n x^n_d)/N \), can be used as the centroid.

Figure 4 plots the frequency distributions of two system call types (i.e., \( D = 2 \)). For now, let us consider only the data points (triangles) on the left-hand side of the graph. The data points are clustered around the star-shaped marker that indicates the centroid of the distribution formed by the points. Now, given a new observation from the monitoring phase, e.g., the point marked ‘A’, a legitimacy test can be devised that tests the likelihood that such an observation is actually part of the expected execution context. This can be done by measuring how far the new observation is from the centroid. Here, the key consideration is on the distance measure for testing legitimacy.

One may use the Euclidean distance between the new observation \( x^* \) and the mean vector of a cluster, i.e., \( ||x^* - \mu|| = \sqrt{(x^* - \mu)^T(x^* - \mu)} \). Although the Euclidean distance (or \( L^2 \)-norm) is simple and straightforward to use, the distance is built on a strong assumption that each coordinate (dimension) contributes equally while computing the distance. In other words, the same amount of differences in \( x_1^* \) and \( x_2^* \) are considered equivalent even if, e.g., a small variation in the usage of system call \( s_2 \) is the stronger indicator of abnormality than system call \( s_1 \). Thus, it is more desirable to allow such a variable contribute more. For this reason, we use the Mahalanobis distance [29], defined as \( \sqrt{(x^* - \mu)^T \Sigma^{-1} (x^* - \mu)} \), for a group of data set \( \mathbf{X} \), where \( \Sigma \) is the covariance matrix of \( \mathbf{X} \). Notice that the existence of \( \Sigma^{-1} \) is the necessary condition to define the Mahalanobis distance; i.e., the difference of the frequency of each system call from the mean (i.e., what is expected) is

\( \Sigma \) is the positive definite. If we set \( \Sigma = I \), the Mahalanobis distance is equivalent to the Euclidean distance. Thus, the Mahalanobis distance is more expressive than the Euclidean distance.
augmented by the inverse of its variance.

Accordingly, if we observe a small variance for certain system calls during the training, e.g., execve or socket, we would expect to see a similar, small, variation in the usage of the system calls during actual executions as well. On the other hand, if the variance of a certain system call type is large, e.g., read or write, the Mahalanobis distance metric gives a small weight to it in order to keep the distance (i.e., abnormality) less sensitive to changes in such system calls. Cluster 2 in Figure 4 shows an example of the advantage of using the Mahalanobis distance over the Euclidean distance. Although it shows an example of the advantage of using the Mahalanobis distance, it is more reasonable to determine that C is an outlier because we have not seen (during the normal executions) frequency distributions such as the one exhibited by C while we have seen a statistically meaningful amount of examples like B. As an extreme case, let us consider D which is quite close to Cluster 3’s center in terms of the Euclidean distance. However, it should be considered malicious because s2 (i.e. the y-axis) should never vary.

Using covariance values also make it possible to learn dependencies among different system call types. For instance, an occurrence of the socket call usually accompanies open and many read or write calls. Thus, we can easily expect that changes in socket's frequency would also lead to variations in the frequencies of open, read and write. Cluster 1 in Figure 4 is such an example that shows covariance between the two system call types. On the other hand, they are independent in Cluster 2 and 3. Thus, using the Mahalanobis distance we can not only learn how many occurrences of each individual system call should exist but also how they should vary together.

Now, given a set of system call distributions, \( X = [x_1, x_2, \ldots, x_N]^T \), we calculate the mean vector, \( \mu \), and the covariance matrix, \( \Sigma \), for this data set. It then can be represented as a single cluster, \( c \), whose centroid is defined as \( (\mu, \Sigma) \). Now, the Mahalanobis distance of a newly observed SCFD, \( x^* \), from the centroid is

\[
\text{dist}(x^*, c) = \sqrt{(x^* - \mu)^T \Sigma^{-1} (x^* - \mu)}.
\]

(1)

If this distance is greater than a cutoff distance \( \theta \), we consider that the execution to be malicious. For example, B in Figure 4 is considered legitimate w.r.t. Cluster 2. One analytic way to derive this threshold, \( \theta \), is to think of the Mahalanobis distance w.r.t. the multinomial normal distribution,

\[
p(x^*) = \sqrt{\det(2\pi \Sigma)^{-1}} \exp \left( -\frac{1}{2} (x^* - \mu)^T \Sigma^{-1} (x^* - \mu) \right).
\]

(2)

That is, we can choose a \( \theta \) such that the p-value under the null hypothesis is less than a significant level \( p_0 \), e.g., 1% or 5%. Appendix explains how to calculate \( \theta \) given a \( p_0 \).

C. Learning Multiple Execution Contexts

In general, an application may show widely varying system call distributions due to multiple execution modes and varying inputs. In such scenarios, finding a single cluster/centroid for the whole set can result in inaccurate models because it would include many non-legitimate points that belong to none of the execution contexts – i.e., the empty space between clusters in Figure 4. Thus, it is more desirable to consider that observations are generated from a set of distinct distributions, each of which corresponds to one or more execution contexts. Then, the legitimacy test for a new observation \( x^* \) is reduced to identifying the most probable cluster that may have generated \( x^* \). If there is no strong evidence that \( x^* \) is a result of an execution corresponding to any cluster then we determine that \( x^* \) is most likely due to malicious execution.

Suppose we collect a training set \( X = [x_1, x_2, \ldots, x_N]^T \) where \( x_n \in \mathbb{R}^D \). To learn the distinct distributions, we use the k-means algorithm [30] to partition the \( N \) data points on a \( D \)-dimensional space into \( k \) clusters. The k-means algorithm works as follows:

1) Initialization: Create \( k \) initial clusters by picking \( k \) random data points from \( X \).
2) Assignment: For each \( x_n \in X \), assign it to the closest cluster \( c(x^*) \), i.e.,

\[
c(x^*) = \arg \min_{c_k \in c} \text{dist}(x^n, c_k).
\]

(3)

3) Update: Re-compute the centroid (i.e., \( \mu \) and \( \Sigma \)) of each cluster based on the new assignments.

The algorithm repeats steps (2) and (3) until the assignments stop changing. Intuitively speaking, the algorithm keeps updating the \( k \) centroids until the total distance of each point \( x_n \) to its cluster,

\[
\text{total-dist}(X, C) = \sum_{n=1}^{N} \text{dist}(x^n, c(x^n)),
\]

(4)

is minimized.

The k-means algorithm requires a strong assumption that we already know \( k \), the number of clusters. However, this assumption does not hold in reality because the number of distinct execution contexts is not known ahead of time. Moreover, the accuracy of the final model heavily depends on the
initial clusters chosen randomly. Hence, we use the **global k-means** method [31] to find the number of clusters as well as the initial assignments that lead to **deterministic accuracy**. Algorithm 1 illustrates the global k-means algorithm. Given a training set $X$ of $N$ system call frequency distributions, the algorithm finds the best number of clusters and assignments. This is an incremental learning algorithm that starts from a single cluster, $c_1$, consisting of the entire data set. In the case of $k = 2$, the algorithm considers each $x^n \in X$ as the initial point for $c_2$ and runs the assignment and updates steps of k-means algorithm. After $N$ trials, we select the final centroids that resulted in the smallest total distance calculated by Eq. (4). These two centroids are then used as the initial points for the two clusters, respectively, in the case of $k = 3$. This procedure repeats until either $k$ reaches a pre-defined $\text{MAX}_k$, the maximum number of clusters, or the total distance value becomes less than the total distance bound $\text{Bound}_{\text{TD}}$. Note that the total distance in Eq. (4) decreases monotonically with the number of clusters. For example, if every point is its own cluster then the total distance is zero since each point itself is the centroid.

The original algorithm assumes the Euclidean distance. As explained above, we use the Mahalanobis distance as in Eq. (1). Meanwhile, k-means($X,C$) (line 11) is the standard k-means algorithm without the random initialization; it assigns the points in $X$ to a $c_k \in C$, update the centroids, repeats until stops, and then returns the clusters with the updated centroids. The standard k-means algorithm uses the Euclidean distance and thus the centroids of the initial clusters are the data points that were picked first. Remember, however, that the Mahalanobis distance requires a covariance matrix. Since there would be only one data point in each initial cluster we use the global covariance matrix of the entire data set $X$ for the initial clusters. After the first iteration, however, the covariance matrix of each cluster is updated using the data points assigned to it.

The clustering algorithm finally assigns each data point in the training set into a cluster. Then, each cluster $c_i \in C$ can be represented by the centroid, $(\mu_i, \Sigma_i)$, that now makes it possible to calculate the Mahalanobis distance of a newly observed SCFD $x^\star$ to each cluster using Eq. (1). The legitimacy test of $x^\star$ is then performed by finding the closest cluster, $c^\star$, using Eq. (3). Thus, if

$$\text{dist}(x^\star, c^\star) = \min_{c_i \in C} \text{dist}(x^\star, c_i) > \theta$$

for a given threshold $\theta$, we determine that the execution does not fall into any of the execution contexts specified by the clusters since $\text{dist}(x^\star, c_i) > \theta$ for all $i = 1, \ldots, k$. We then consider the execution to be malicious. As an example, for the new observation $C$ in Figure 4, Cluster 2 is the closest one and $C$ is outside its cutoff distance. Thus, we consider that $C$ is malicious. Note that, as shown in the figure, the same cutoff distance defines different ellipsoids for different clusters; each ellipsoid is a equidistant line from the mean vector measured in terms of the Mahalanobis distance. Thus, a cluster with small variances (i.e., less varying execution context) would have a smaller ellipsoid in the Euclidean space.

**D. Reduced SCFD**

The number of system call types, i.e., $D$, is quite large in general. Thus, the matrix calculations in Eq. (1) might result in an unacceptable amount of analysis overhead. However, embedded applications normally use a limited subset of system calls. Furthermore, we can significantly reduce the dimensionality by ignoring system call types that never vary. Consider Cluster 3 from Figure 4. Here, $x_2$ can be ignored since we can reasonably expect it to never vary during the normal execution. Thus, before running the clustering algorithm, we reduce $S$ to $S' = \{s_{d_1}, s_{d_2}, \ldots, s_{D'}\}$, where $D' \leq D$, such that the variance of $x_d$ for each $s_d \in S'$ is non-zero in the entire training set $X$. However, we should still be able to detect any changes in such system calls that never varied (including those that never appeared). Thus, we merge all such $x_d$ in $S-S'$; the sum should not change in normal executions. In case $D'$ is still large, one may apply a statistical dimensionality reduction technique such as Principal Component Analysis (PCA) [32].

Suppose the reduced dimension of SCFD is $D'$ and we have learned $K$ clusters. To store the cluster information (i.e., $(\mu_i, \Sigma_i)$), we need a memory of $KD'$ elements and $KD'^2$ elements to store the mean vectors and the covariance matrices, respectively, where each element stores a floating-point data.

**IV. Evaluation Framework**

In this section, we present the implementation details for our prototype.

**A. Target Application**

We use **Motion** [24], an open-source program that monitors images captured from a camera and detects motion by tracking changes between image frames as illustrated in Figure 5. It is often used for surveillance purpose and provides live streaming and external program execution when certain events (e.g., motion detection, on file creation, etc.) are detected.

Figure 5 also shows **Motion**’s main execution process. The main loop consists of a series of blocks. Each loop starts by capturing an image frame from the camera using the Video4Linux (V4L) interface. Next, motion detection algorithm looks for changes from the previous frames. When a change (i.e., motion) 3Note that $\mu$ and $\Sigma$ values are calculated from the clustering algorithm which is an offline analysis. Thus we store $\Sigma^{-1}$ for computational efficiency.

4In fact, $s_2$ cannot be ignored in the example depicted in Figure 4 since its variance is non-zero in clusters 1 and 2.
is detected, each frame is saved to the filesystem. Following this, some pre-defined event actions could trigger external programs (such as executing a script file, uploading image to a remote image server, etc.). This main loop repeats at the specified frame rate (such as 3 frames per second). Depending on the events, some of the blocks may not execute in every loop. In our configuration, a python script that logs the current time in a file executes (by on_motion_detected event handler) each time a motion is detected, and the wput Linux command is executed to upload the newly created images (by on_picture_save event handler) to a remote server. These external commands are executed by separate processes forked by the main process.

B. System Implementation

We implemented a prototype of our SCFD-based anomaly detection system on a Raspberry Pi 2 Model B board [23]. It has a quad-core ARM Cortex-A7 CPU. Each core runs at 900 MHz. The system has a memory of 1 GB and runs Linux 3.18.

Figure 6 shows our system call monitoring framework implemented on Broadcom BCM2836 SoC (System-on-Chip) [33] on the Raspberry Pi 2 board. We inserted a hook in the software interrupt handler that dispatches each system call handler. The hook sends the system call number and the PID (Process ID) of the caller to the monitoring process (called Secure Monitor) on Core 3 through a set of mailboxes. Each mailbox on BCM2836 SoC is a 32-bit wide core-to-core communication channel. The secure monitor performs the detection process presented in Section III using the SCFD built from the reported information.5 We implemented the secure monitor as a baremetal application for the purposes of our proof-of-concept. We created a Linux kernel module that tells (through another mailbox) the secure monitor what processes to monitor.

Since we collect system call usage information at the operating system layer (i.e., software interrupt handler), the OS is our trust-base. Note that we could either run the secure monitor inside or on top of the OS as done by most system call monitoring/auditing modules [34], [35]. One can add more security by utilizing a hardware-supported partitioning mechanism, e.g., the ARM TrustZone [27], and protecting the security monitor even if the main system is compromised.

C. Attack Scenarios

Considering the purpose and the functionality of Motion, the primary security concerns are privacy and availability.

Hence, we consider the following attack scenarios:

1) Attack 1: One attack is the leaking of images captured by Motion while leaving the original functionality intact. We consider the case where an attacker saves the images at a desired location in the filesystem with the intention that the collection will be used/retrieved later.

2) Attack 2: The attacker corrupts the images captured from the camera so that no motion can be detected. Specifically, the attacker erases frame(s) by calling memset. Note that this attack does not require any system calls.

The attacker tries to implement the above attacks as simply as possible (e.g., using existing routines/libraries) because otherwise the system call usage will diverge in an obvious way. Note that we consider only the cases that change the system call usage of Motion directly or indirectly. If the attacker say, had higher privileges in the system then he could perform more active attacks such as killing the Motion process, copying the legitimately-saved images out of the device, deleting files, disabling the camera, etc. Such attempts can be detected by other techniques. Also, we do not make any assumptions as to how the compromised program is present on the device. The modified program may have already been installed or the user may have downloaded the modified (open) source code or the executable binary.

V. Evaluation Results

We now evaluate the SCFD method on the prototype described in the previous section. We obtained a training set that consists of 2420 loop executions of Motion that ran under normal conditions (i.e., no attack present) for about 15 minutes. Motion used total 15 types of system calls.

A. Sequence-based Security Analysis

We first show that it is feasible to implement the attack scenarios described in Section IV-C while avoiding detection from sequence-based approaches.

Figure 7 summarizes the system call sequences made in each loop in normal situations.6 Each loop always starts by storing the current times (for time keeping) and taking the current image frame from the camera. Then, if motion is detected, a separate process is forked to execute an external command upon the on_motion_detected event and the image frame is saved as a file. If the time to take a snapshot arrives (e.g., once every 5 seconds, for archiving purposes) the resulting image is saved as a file. Next, the loop feeds an image to a webcam-client if any are viewing. This if block generates several variations that cannot be represented by a single sequence. The loop then ends with the frame-rate control. The shortest sequence is when all the if conditions are false.

The three if blocks are independent; each loop may execute only one, a pair, or all of them depending on the current situation. This creates various execution contexts. The system call usages can vary further when images are saved to files, as can be seen from the first two if blocks. This is due to the

5We created a custom system call that indicates an execution boundary. We inserted a special system call at the end of Motion’s main loop. The secure monitor recognizes this system call as the end of one execution.

6The sequence shown in the figure is the most representative one. In some loops, for example, the first ioctl in the second line may skip, and nanosleep in the last line may not be called if there is no need to insert a delay to meet the next frame time.
Motion loop {

gettimeofday-gettimeofday

(ioctl)-rt_sigprocmask-ioctl-rt_sigprocmask /* Frame Capture */

If (motion_detected) /* Run external command upon 'on_motion_detected' event*/

close

open-fstat64-mmap2-write-...-write-close-mmap-clone-write /* Save frame image*/
write

} /* write chain. Length depends on image size. */

If (time to take a snapshot) {

open-fstat64-mmap2-write-...-write-close-mmap-clone-write /* Save snapshot image*/
unlink-symlink /* Update symbolic link to the latest snapshot file*/

}

select /* Wait for a webcam-client */

If (a webcam-client is waiting) {

/* Several variations are made with the these calls*/

(accept-ioctl-write-(write-munmap-close)-(mmap2-gettimeofday)-(write)-(write)-(munmap)

}

gettimeofday-(nanosleep) /* Frame-rate Control */
}

Fig. 7. The system call sequences made by Motion in each loop. Total 15 different types of system calls are used. The three if blocks can independently execute and thus create various execution contexts. Further variations are made by the write chain when saving images to files. The parenthesized calls (e.g., (ioctl)) may sometime present/skip.

varying length of the write chain that depends on the image size.

We now explain how the attacks can be carried out while avoiding detection from sequence-based approaches.

**Attack 1:** Notice, from Figure 7, the system call sequence that is made when writing to files:

open-fstat64-mmap2-write-...-write-close-mmap-clone-write

It is generated by a common routine, put_picture, in Motion. Hence, the attacker can use the very same function to save the current frame image at a desired location by inserting the following small piece of code:

```c
const char* org_filepath = cnt->conf.filepath;
cnt->conf.filepath = "*/path/to/attackers_desired_location*";
event(cnt, EVENT_IMAGE_DETECTED, cnt->imgs.image_ring[|nt->
imgs.image_ring_out].image, NULL, NULL, &cnt->imgs.image
ring[|nt->imgs.image_ring_out].timestamp_tm);
cnt->conf.filepath = org_filepath;
```

The event function above is identical to what is called by the original code (in the first two if blocks in Figure 7) and it in turn calls the put_picture library routine. The attacker only needs to change the path to store the image (i.e., cnt->conf.filepath) in the configuration and restore it back before and after calling the event function, respectively.

```c
If (motion_detected) {

clone

open-fstat64-mmap2-write-...-write-close-mmap-clone-write
write

} /* Inserted by attacker*/

If (time to take a snapshot) {

open-fstat64-mmap2-write-...-write-close-mmap-clone-write
unlink-symlink

}
```

Fig. 8. Attacker can insert a simple piece of code that uses the same routine as the legitimate code calls to save an image data to the filesystem. If the code is inserted as depicted, the resulting system call sequences would look normal, as depicted.

Now, the attacker can place this piece of code (followed by a bogus write call) between the two file write operations, as depicted in Figure 8. The figure shows the case when the attacker wants to steal every motion frame. The attacker can instead steal snapshots by inserting the code at the beginning of the second if block (i.e., before C).

Now, it is difficult for sequence-based approaches to catch this attack because the resulting subsequences look normal. Let us consider the following cases:

1) Only the first if block executes (Case 2 in Figure 9): First of all, any subsequences made only by the inserted code itself (marked as B in Figure 8) are same with those made by the legitimate code (marked as A). Now, the subsequences that span the legitimate (A) and the inserted codes (B) are same with those made by the transition from A to C when the attack code was not inserted (Case 3) in Figure 9 which is a normal sequence). That is, the execution looks as if both frame image and snapshot are saved (i.e., both if blocks execute) although the second if block did not execute. Also, the subsequences that include the tail of B look identical to those of A when only A executes (Case 1 in Figure 9). Hence, they are legitimate too.

2) Both if blocks execute (Case 4 in Figure 9): Now suppose the inserted code executes between the two file write operations, as depicted in Case (4) of Figure 9. Similar to the situation above, the subsequences generated by the transitions from A to B and B to C are still legitimate with respect to the patterns that can be learned from the normal case (i.e., Case 3). This is because what A and B generate together (that is, patterns that span across A and B) do not look different from what A and C would generate, and for the same reason, the patterns generated by B and C look normal too.

In the same way, the attacker can even execute an arbitrary external command too by changing the configuration parameter that stores the string of external command that will execute upon image file creation. Hence, the attacker can upload the leaked image to a remote server by setting Motion parameter to externally execute wput Linux command.
normal executions. Next, the length of the write chain when saving the snapshot image to a file changes. This is because the erased frame produces 14KB of green images. The reduced size of the images result in shortened write chain as depicted in Case 2 of Figure 10. A single write call writes 4KB of data to the file and hence the write chain’s length becomes 4.

It is significantly shorter than a normal length which is longer than 70 in general.

Sequence-based methods may or may not be able to detect such anomalies, depending on the length of patterns they learn.9 If we have learned the patterns of length at most 5, the subsequences are still legitimate with respect to the normal subsequences, as can be seen from Case 2 of Figure 10. On the other hand, if we have learned longer (e.g., at least 6) patterns, we can know that mmap2-write-write-write-write should be followed by another write in normal situations (as in Case 1 of the figure). The abnormal situation illustrated in Case 3 is thus malicious with respect to such normal subsequences. Note that the difficulty of capturing such an abnormal situation stems from the long chain of write calls in the normal execution scenarios. Hence, it is not straightforward for sequence-based methods to learn the relationship between mmap2 and close especially because the write chain’s length can vary with data as well. Note also that, if the image size got larger (say, by writing random values to image frame) instead and thus made the write chain longer than usual, sequence-based methods cannot detect this behavior because the only change would be that there are more subsequences that consists only of write that have a legitimate length.

B. SCFD Training

With the training set obtained from the system under normal conditions, we applied the SCFD learning algorithm presented in Section III. Out of 15 types of system calls used by Motion, two types, select and rt_sigprocmask, showed zero variance. As shown in Figure 7, these two system calls always execute once and twice, respectively, in each loop. Hence, the algorithm first reduces the dimensionality of SCFDs to 13.

Figure 11 visualizes the training result obtained with settings \( \text{MAX}_{x} = 20 \) and \( \text{Bound}_{D} = 1000 \). The table in the middle is the training set (only unique SCFDs are shown) and the ones around it are the resulting clusters. Each row represents an SCFD and the colors represent high (orange color) and low (green color) counts for each system call type. As can be seen from the result, the 2420 SCFDs are clustered into 11 clusters. From observing the resulting clusters, we find the following execution contexts:

1) Cluster 1 represents the case when no event occurs during a loop shown in Figure 7. The loop only takes the current image frame and none of the if blocks in Figure 7 execute.

8No matter how patterns are learned. They could be learned by the fixed-length method such as N-gram [13] or variable-length such as Markovian model (VMM) [36] or Probabilistic Suffix Tree [17], etc.

9Again, we do not make any assumptions on the way that the sequence patterns are learned. It could be a fixed-length or variable-length.
Clusters 2, 4 and 5 are also the cases when no images are saved to files, because the related system calls (e.g., open, fstat64, close) do not appear and also the number of write calls is few. The differences among the three clusters are due to the last if block (i.e., webcam remote view-related) in Figure 7 which shows several variations.

Clusters 3, 6, 8, 10, and 11 correspond to the executions that write an image file once, because the file-related system calls appear just once per SCFD (i.e., per loop). In addition, the write calls are used accordingly. Among them, Clusters 6, 10, and 11 write snapshot images (i.e., the second if block in Figure 7). Cluster 11 is when an image is fed to a webcam-client as more mmap and unmap are observed. The only difference between Clusters 6 and 10 is the number of write calls; it is fixed to 74 in Cluster 6, while Cluster 10 has everything but 74.

The fewer number of unlink and symlink in Clusters 3 and 8 (than 6, 10, and 11) suggest that these two correspond to the executions that write frame images (i.e., the first if block). Also, clone should be called twice in that case.

Cluster 9 corresponds to the case when both, the motion frame and snapshot files are saved (because of the reasons explained above). This cluster covers both the cases when an image is fed or not fed to webcam-client. Increasing the number of clusters will split the two cases.

Cluster 7 is a mixture of some rare SCFDs that are similar to other clusters but vary in a very small way (due to the last if block in Figure 7). Such differences caused them to stand out in comparison to other clusters. Each one was also not representative enough to create its own cluster. For example, only 4 out of 2420 SCFDs in the training set had a non-zero count of accept and these are assigned to Cluster 7. This cluster can be split into smaller ones if we increased the number of clusters.

Overall, the results show the changes in the execution contexts as various events occur individually or together.

C. Accuracy

Figure 12 shows the closest cluster for each SCFD (for 300 SCFDs obtained during a normal situation) and the corresponding execution context. The shaded areas represent the time period when motion is detected – during which a frame image is saved to a file. We can also see that a snapshot is saved at regular intervals (every 5 sec) regardless of motion detection. Overall, the results show the changes in the execution contexts as various events occur individually or together.

1) Attack 1: We inserted the code block that leaks out the current frame image to the attacker’s desired location (code block “B” as shown in Figure 8) and then obtained a test set of 1003 SCFDs. Note that not all of them include the attack because the inserted code executes only when a motion is detected. 603 out of the 1003 SCFDs correspond to the case that did not detect a motion and thus are normal.

The rest of the SCFDs can be divided into two groups as shown in Figure 13. The first group (upper-right) looks very similar to the ones in Cluster 9 (in Figure 11) that saves both motion frame and snapshot images. If the test SCFDs were legitimate, then they should have used unlink and symlink system calls once as shown in the Motion’s normal system call usage in Figure 7. Since the test SCFDs did not use the calls, they are classified as abnormal with respect to the learned patterns. Of course, the attacker could insert bogus unlink and symlink for this particular execution context. However, then the resulting sequences are identical to those made by the normal code (when both images are saved) and no system call-based detection methods can differentiate the two cases, which does not fall...
The second group (at bottom-right in Figure 13) consists of SCFDs observed when the inserted code executes between the two legitimate file operations (see Figure 8). The resulting SCFDs are clearly abnormal as there are three file operations and hence too many write calls.

2) Attack 2: This attack does not use any system calls; it just changes the values of the data (i.e., image). As explained earlier, this attack produces 14KB of frame images, which results in shortened write chains as depicted in Figure 10. Hence, calls to write is much less frequent when compared to normal executions.

The SCFDs shown above, obtained when Attack 2 is enabled, are quite close to Cluster 6 (top) and Cluster 11 (bottom), respectively. However, these SCFDs are always classified to be abnormal because the image sizes (due to the number of write calls) are not typical when saving snapshot files during normal executions. The attacker could have circumvented our detection method if, say the frame image is just replaced with another that has a similar size as the unmodified frame images. However, again, such case is out of scope of our threat model because the system call usage does not change.

The false positive rate is just as important as the detection rate because frequent false alarms degrade system availability. To measure the false positive rates, we obtained a new set of SCFDs by running the system without activating any attacks and measured how many times the secure monitor classifies an execution as being abnormal. For the cut-off distance $\theta$ with $p_0 = 5\%$, 4 out of 1755 executions (0.23%) were classified as malicious. With $p_0 = 1\%$, i.e., a farther cut-off distance, it was reduced to just 1 (0.06%). Such a lower significant level relaxes the cutoff distance and produces fewer false alarms because even some rarely-seen data points are considered normal. However, this may result in lower detection rates as well. In the attack scenarios listed above, however, the results did not change even with the lower significant level. This is a consideration for system designers to take into account when implementing our detection methods; they will have a better feel for when certain executions are normal and when some are not. Hence, they can decide to adjust values for $p_0$ based on the actual system(s) being monitored.

In Section V-A, we showed that sequence-based approaches may fail to detect abnormal deviations in situations that naturally have a high-level variance in the execution contexts or use data that is very diverse. Such instances require a global view on the frequencies of different system call types made during the entire execution and the correlations among different types. Sequence-based approaches are sensitive to local, temporal variations, e.g., an unusual transition from one system call to another. Our SCFD might not catch such a small, local variation. Hence, one can use these two approaches together to improve the overall accuracy of the system call-based anomaly detection for embedded systems.

While it is true that the accuracy of the method may depend on the attacks that are launched against the system, in reality an attacker would need to not only know the exact distributions of system call frequencies but also be able to implement an attack with such a limited set of calls -- both of these requirements significantly raise the difficulty levels for would-be attackers.

D. Time Complexity

To evaluate the time complexity of the proposed detection method, we measured the times to perform the analysis. The times are measured from the moment when a new observation is given until the closest cluster is found (Eq. (3)). We tested for different configurations of the SCFD dimensionality and the number of clusters. The statistic is based on 10000 samples per configuration collected on our prototype system.

As Table I shows, the detection process is fast. This is possible because we store $\Sigma^{-1}$, the inverse of the covariance matrix, of each cluster, not $\Sigma$. A Mahalanobis distance is calculated in $O(D^2)$, where $D$ is the number of system call types being monitored (i.e., SCFD dimensionality), since in $(x^* - \mu)^T \Sigma^{-1} (x^* - \mu)$, the first multiplication takes $O(D^2)$ and the second one takes $O(D)$. The results (top to bottom) in the table above show such a trend. Note that it would have taken $O(D^3)$ if we stored the covariance matrix itself instead of its inverse; since a $D \times D$ matrix inversion takes $O(D^3)$. We can also see from the results that the analysis time increases linearly with the number of clusters.

More importantly, the time complexity of our method is independent of how often and many times the application uses system calls; it only depends on the number of system call types being monitored. This is determined in the training phase and does not change during the monitoring phase (see Section

TABLE I

<table>
<thead>
<tr>
<th>SCFD Dimension</th>
<th>5 Clusters</th>
<th>10 Clusters</th>
<th>15 Clusters</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>4.710 $\mu$s (0.522 $\mu$s)</td>
<td>8.348 $\mu$s (0.545 $\mu$s)</td>
<td>11.843 $\mu$s (0.373 $\mu$s)</td>
</tr>
<tr>
<td>10</td>
<td>11.262 $\mu$s (0.501 $\mu$s)</td>
<td>21.318 $\mu$s (0.470 $\mu$s)</td>
<td>31.474 $\mu$s (0.503 $\mu$s)</td>
</tr>
<tr>
<td>13</td>
<td>16.306 $\mu$s (0.463 $\mu$s)</td>
<td>31.582 $\mu$s (0.501 $\mu$s)</td>
<td>46.859 $\mu$s (0.356 $\mu$s)</td>
</tr>
</tbody>
</table>

Note that we found 11 clusters from the training. To test for 15 clusters, we simply added 4 duplicated clusters (i.e., means and inverse covariance matrices).
III-D). On the other hand, the overheads of sequence-based approaches are highly dependent on the application complexity (i.e., how many system calls are made). Hence, the SCFD method has a deterministic time complexity.

E. Limitations and Discussion

One of the limitations of our detection algorithm is that it checks for anomalies after execution is complete (for each invocation). Combining a sequence-based method with our SCFD can be a solution if such attacks can be detectable by the former. If not, one can increase the chances of detection such problems by splitting the whole execution range into blocks [19] and checking for the distribution of system calls made in each block as soon as the execution passes each block boundary. This also can relax the assumption of repetitive execution of the target application (explained in Section II) because an analysis is applied at block-level. This, however, would need more computation in the secure monitor at run-time, more storage for profiles and a few more code modifications.

Another way to handle this problem is to combine this analysis/detection with other behavioral signals, especially ones that have a finer granularity of checks, e.g., timing [19]. Since some blocks may use very few system calls or even a very stable subset of such calls we can monitor the execution time spent in such a block to reduce the SCFD-based overheads (which is still low). This keeps the profile from bloating and prevents the system from having to carry out the legitimacy tests. We can also use the timing information in conjunction with the system call distribution, i.e., by learning the normal time to execute a distribution of system calls, we can enforce a policy where each application block executes all of its system calls within (fairly) tight ranges. This is, of course, provided that the system calls do not show unpredictable timing behavior. This makes it much harder for an attacker who imitates system calls [37] or one that replaces certain system calls with malicious functions [25].

We can model a primitive operation (such as a network activity, a file operation, etc.) as a topic and then represent an execution context as a mixture of several primitive operations [38]. In this paper, we build instead a pragmatic lightweight module. One of the main drawbacks of the k-means clustering algorithm is that one may need to know or pre-define the number of clusters. That is, system behaviors should be correctly represented by k numbers of multinomial (Gaussian) distributions of histogram. Some large-scale systems would have many heterogeneous modes (distributions). In this case, the appropriate solutions would be using non-parametric topic models such as Dirichlet process. However, we empirically observed that many embedded systems with predictable behavior can be represented by a tractable number of clusters. Thus, we used a simpler model with the k-means cluster.

VI. RELATED WORK

Forrest et al. [10] build a database of look-ahead pairs of system calls; for each system call type, what is the next nth system call for i = 1, 2, up to N. Then, given a longer sequence, the percentage of mismatches is used as the metric to determine abnormality. Hofmeyr et al. [13] extends the method by profiling unique sequences of fixed length N, called an N-gram, to reduce the database size. The legitimacy test for a given sequence of length N is carried out by calculating the smallest Hamming distance between it and the all sequences in the database. The N-gram model requires a prior assumption on suitable N because it affects the accuracy as well as the database size. Marceau [15] proposes a finite state machine (FSM) based prediction model to relax these requirements and Eskin et al. [16] further improves by employing a wild-card for compact sequence representation. Markovian techniques such as Hidden Markov model (HMM) [14] and variable-order Markov chain [17] have also been explored. A similar approach to our work is [12], in which the system call counts of Android applications (traced by a software tool called strace) are used to find malicious apps. Using a crowdsourcing, the approach collects the system call counts of a particular application from multiple users and applies k-means (with Euclidean distance metric) to divide them into two clusters. A smaller cluster is considered to be malicious based on the assumption that benign apps are the majority.

There has also been work on system call arguments monitoring. Mutz et al. [11] introduce several techniques to test anomalies in argument lengths, character distribution, argument grammar, etc. Maggi et al. [39] use a clustering algorithm to group system call invocations that have similar arguments.

The usual way of system call instrumentation relies on an audit module in the OS layer. Hardware-based system call monitoring mechanism can improve the overall security of the system by cutting off a potential vulnerability – the software audit module. Pfoh et al., [40] proposed Nitro, a hardware-based system call tracing system where system calls made inside virtual machines. We note that our detection method (Section III) is orthogonal to how system calls are traced. Hence we can implement it on systems like Nitro. Other types of instrumentation include static analysis of program source code [41] and user-level processes for system call interposition [42].

The SecureCore architecture [19] takes advantage of the redundancy of a multicore processor; a secure core is used to monitor the run-time execution behavior of target applications running on a monitored core. The original architecture is designed to watch applications timing behavior. The authors extended the architecture to monitor memory behavior for system-wide anomaly detection [21] and to protect in-place security monitoring module [28].

VII. CONCLUSION

In this paper we presented a lightweight anomaly detection method that uses application execution contexts learned from system call frequency distributions of embedded applications. We demonstrated our technique for a real-world open-source application and showed that the proposed detection mechanism could effectively complement sequence-based approaches by detecting anomalous behavior due to changes in high-level execution contexts. We plan to improve the learning and analysis methods using the topic modeling approach (explained in Section V-E) to deal with large-scale heterogeneous behaviors of complex embedded applications.

REFERENCES


In general, there is no analytic solution for calculating the cumulative distribution function (CDF) for multivariate normal distributions. However, it is possible to derive the CDF with Mahalanobis distance. The cutoff distance \( \theta \) can be derived by finding the smallest distance that makes the probability that a data point \( x \), which in fact belongs to the cluster and has a distance farther than \( \theta \), is not greater than \( p_0 = 0.01 \) or 0.05.

First, let \( z \) be a Mahalanobis distance from a multivariate normal distribution. Then,

\[
\int_0^\theta c \cdot e^{-\frac{1}{2}z^2} \, dz = 1 - p_0, \tag{5}
\]

where \( c \) is a normalizing constant that satisfies Eq. (5) with \( \theta = \infty \) and \( p_0 = 0 \) by the definition of a probability density function. This results in \( c = 1/1.25331 \)

\[
\int_0^\infty e^{-\frac{1}{2}z^2} \, dz = 2\sqrt{\pi} = 2.53316 \\
\int_0^\infty e^{-\frac{1}{2}z^2} \, dz = 2\sqrt{\pi} = 2.53316
\]

where \( \text{erf}(z) \) is the error function and is 1 and 0 for \( z = \infty \) and \( z = 0 \), respectively. Accordingly, Eq. (5) becomes

\[
\frac{1}{1.25331} \int_0^\theta e^{-\frac{1}{2}z^2} \, dz = \frac{1}{1.25331} \left[ 1.25331 \cdot \text{erf}(0.707107 \cdot z) \right]_0^\theta = \text{erf}(0.707107 \cdot \theta) = 1 - p_0.
\]

Therefore, the cutoff distance \( \theta \) for a significant level \( p_0 \) is

\[
\theta = \frac{\text{erf}^{-1}(1 - p_0)}{0.707107}. \tag{6}
\]

For \( p_0 = 1\% \) and 5\%, \( \theta \approx 2.57583 \) and 1.95996, respectively. The cutoff distance is not bounded (i.e., \( \theta = \infty \)) when \( p_0 = 0\% \) and is 0 when \( p_0 = 100\% \).