Intrusion Detection Using Execution Contexts Learned from System Call Distributions of Real-Time Embedded Systems

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
Email: {mkyoon, †sibin, lrs}@illinois.edu, ‡jaesik@unist.ac.kr, §mihai@qti.qualcomm.com

Abstract—The increasing number of security threats faced by real-time embedded systems requires the development of effective intrusion detection mechanisms. However, the limited resources in such systems, viz., computational power and storage, prevent current techniques from being effective. In this paper, we propose a lightweight method with a deterministic time complexity for detecting anomalous executions using a distribution of system call frequencies, i.e., the number of occurrences of each system call type during execution. We use the global k-means clustering with the Mahalanobis distance to learn the legitimate execution contexts of real-time embedded applications and then monitor them at run-time to detect intrusions. We also present minor processor modifications to aid in this process. Initial prototypes show that the proposed method can effectively detect anomalous executions without relying on overly sophisticated analyses. It can also detect certain attacks that current methods fail to do so.

I. INTRODUCTION

An increasing number of attacks are targeting real-time embedded systems [1], [2]. Their main aim is to compromise the security, and hence safety, of such systems. The ever-growing complexity of modern real-time applications exposes more security flaws [3]. It is not an easy task to retrofit real-time systems (RTS) with security mechanisms that were developed for more general purpose scenarios since RTS (a) are often constrained in processing power, memory, battery life, etc. and (b) must meet stringent timing requirements. On the other hand, these very properties of RTS also make them amenable to using certain security mechanisms [4]. The regularity in their execution means that we can detect intrusions by monitoring the behavior of such applications; deviations from the expected behavior can be considered to be malicious since the set of what constitutes legitimate behavior is limited by design.

Traditional behavior-based intrusion detection systems (IDS) rely on specific signals such as network traffic [5]. Hence, in this paper, we present an intrusion detection mechanism for real-time embedded systems using a system call frequency distribution (SCFD). Figure 1 shows an example SCFD obtained from an application described in Section IV. It is a vector of non-negative integers where each entry represents the number of occurrences of a particular system call type for each execution run of an application. The key advantage of this SCFD-based approach over sequence-based ones is that we can learn the high-level execution context by looking at how different types of system calls are distributed and correlated. The sequence-based approaches rather profile the local, temporal relations among system calls within a limited time frame.

Figure 2 highlights the difference: the left side shows the systems calls made when an image is being uploaded to an FTP server – this shows repetitive behavior. A smart attacker...
may use the very same routine to circumvent the process and upload the image to its own server right after the normal operation completes, as shown on the right side. Existing sequence-based methods cannot detect this extra network activity since the system call subsequences at the point of activation of the malicious code (and even when it actually takes place, e.g., write-write-write-socket) do not differ from what can be observed during normal operations (shown at the top-right corner). Our SCFD-based method, as we show in this paper, can easily detect this situation by analyzing the frequencies of the calls. Also, if the attacker corrupts the integrity of the data (for instance, changes the image size to downgrade its quality) then our methods are able to detect such problems – this is not easy for existing sequence-based methods. Moreover, the sequence-based methods invoke an analysis each time a call is made. This could be problematic if the applications under monitoring make a lot of system calls, thus resulting in significant (potentially unbounded) overheads at run-time. However, our approach has a bounded time complexity in that the analysis is independent of how often and many times the applications use system calls since we analyze each execution, i.e., an SCFD, as a whole.

The application(s) we monitor, however, may exhibit multiple execution contexts due to different operating modes and/or inputs. Representing such variations by a single behavioral context can lead to inaccuracies in the model due to the smoothing out of irregularities. Hence, we use a cluster analysis to find distinct execution contexts from a set of SCFDs. When checking for the legitimacy of an execution run, the most probable execution context is found from a set of clusters using a similarity metric. If the similarity is not statistically meaningful then the execution is considered malicious. In Section III we explain how to model the regularity of system calls and measure the similarity.

Our detection method is lightweight and thus fits well for real-time embedded applications. This is due to the coarse-grained, concise representation of the application behavior as explained above. It can be implemented either at the operating system layer [20] or even used as an offline analysis mechanism. However, for added security, we demonstrate an implementation on the SecureCore architecture [14]. Minor modifications of the instruction set architecture (ISA) of a modern multicore processor coupled with some other hardware changes enables us to monitor and analyze the run-time system call usage of applications in a secure, non-intrusive manner. Section IV presents the hardware modifications needed to facilitate this on-line system call monitoring and intrusion detection.

We implemented our prototype on a full-system simulator [21]. The experimental results, based on an example application and various attack scenarios, show that SCFDs can effectively detect abnormal execution behavior in real-time embedded systems; the results also show that the methods in this paper are able to detect attacks that are difficult for sequence-based approaches. Detailed results including a comparison with an existing sequence-based technique is presented in Section IV.

Hence, the high level contributions of this paper are:
1) we developed a lightweight method with a deterministic
   technique is presented in Section IV.
time complexity for detecting anomalous behavior of real-time embedded systems based on the distribution of system call frequencies (Section II).

2) we provide an architectural solution for secure, non-intrusive monitoring and analysis of system behavior (Section III).

3) we demonstrate our techniques using a prototype of the intrusion detection system and evaluate it using various attack scenarios including a real attack (Section IV).

A. Assumptions and Adversary Model

The following assumptions are made without loss of generality: (i) We consider a real-time embedded application that executes in a periodic fashion. We monitor and perform a legitimacy test at the end of each invocation of a task. (ii) Most of the possible execution contexts can be profiled ahead of time – this is a general requirement of most behavior-based IDS. This can be justified by the fact that most real-time embedded applications have a limited set of execution modes, input data fall within fairly narrow ranges, and they use a limited subset of system calls. Also, a significant amount of analysis of real-time systems is carried out post-design/implementation anyways for a variety of reasons (to guarantee predictable behavior for instance). Hence, this information about the usage of system calls can be rolled into this a-priori analysis. (iii) The initial state of the application is trustworthy. The profiling is carried out prior to system deployment. Also, any updates to the applications must be accompanied by a repeat of the profiling process. The application(s) could be compromised after the profiling stage, but we assume that the stored profile(s) cannot be tampered with. Again, such (repeat) analysis is typical in such systems – anytime the system receives updates. (iv) We consider the threat model that involves changes to the behavior of system call usage. If an attack does not invoke any system calls (e.g., tainting a data on memory), the activity at least has to affect executions afterward so that the future system call distribution may change. The methods in this paper, as they stand, cannot detect attacks that never alter system call usage and that just replace certain system calls by hijacking them (e.g., altering kernel system call table) [22]. (v) We consider malicious code that can be secretly embedded in the application, either by remote attacks or during upgrades. The malicious code activates itself at some point after system initialization. We are not directly concerned with how the malicious code gained entry, but focus more on what happens after that.

II. INTRUSION DETECTION USING EXECUTION CONTEXTS LEARNED FROM SYSTEM CALL DISTRIBUTIONS

We now present our novel methods to detect intrusions in real-time embedded applications by monitoring changes of system call frequencies. The main idea is to learn the normal system call profiles, i.e., patterns in system call frequency distributions, collected during legitimate executions of a sanitized system. Once the system is deployed, we observe the system call usage of the applications at run-time. If the run-time behavior deviates from profiles obtained during the analyses of the normal executions, then we claim that the application/system has been infected with a malware.

Analyzing profiles is challenging especially when such profiles change, often dramatically, depending on the execution contexts. We address this issue by clustering the distribution of system calls from legitimate behavior. Each cluster then can be a signature representing an execution context, either in a specific mode or for similar input data. Then, given a new observation at run-time, we test how similar it is to each previously calculated cluster. If there is no strong evidence that it is a result of a specific execution context then we consider the execution to be malicious. In what follows, we provide details on these methods.

A. Definitions

Let \( S = \{s_1, s_2, \ldots, s_D\} \) 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 \( S \). 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, \( x^n = [x^n_1, x^n_2, \ldots, x^n_D]^T \).

Next, 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 \( X = [x^1, x^2, \ldots, x^N]^T \). The clustering algorithm (Section III-C) then maps each \( x^n \in N^D \) to a cluster \( c_1 \in C = \{c_1, c_2, \ldots, c_k\} \). We denote \( c(x^n) = c_1 \) if \( x^n \) is in cluster \( c_1 \). Our algorithm uses the Mahalanobis distance metric \([23]\), which will be explained in Section III-B, to measure how similar \( x^n \) to each \( c_1 \). We denote it as \( dist(x^n, c_1) \).

B. Intrusion Detection for 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 the centroid, and make a small variation from it according to, for example, input data or execution flow. This is a valid assumption for real-time embedded systems since the code in such system tends to be fairly limited in what it can do. Hence, such analysis is quite powerful in detecting variations and hence, catching intrusions.

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. For instance, consider Figure 3 that plots the frequency distributions of two system call types (i.e., \( D = 2 \)). For now, let us consider only the data points (triangles) at the bottom left corner 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 legitimate 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.

One may, of course, 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 \( x_d^a \) and \( x_d^b \) are considered equivalent even if, e.g., a small variation in the usage of system call \( d_2 \) is the stronger indicator of abnormality than system call \( d_1 \). Thus, it is more desirable to allow such a variable contribute more. For this reason, we use the Mahalanobis distance \(^{23}\), defined as:

\[
dist_M(x^*, X) = \sqrt{(x^* - \mu)^T \Sigma^{-1}(x^* - \mu)},
\]

for a given group of data set \( X \), where \( \Sigma \) is the covariance matrix of \( X \). 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 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 such values. Cluster 2 in Figure 3 shows an example of the advantage of using the Mahalanobis distance over the Euclidean distance. Although \( C \) is closer to the centroid than \( B \) in terms of the Euclidean distance, it is more reasonable to determine that \( C \) is an outlier because we have not seen (during the profiling phase) 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 \( E \) which is quite close to Cluster 4’s center in terms of the Euclidean distance. However, it should be considered malicious because \( s_2 \) should never vary.

Using covariance values also make it possible to learn dependencies among 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 3 is such an example that shows covariances between the two system call types. On the other hand, they are independent in Cluster 2, 3 and 4. 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, \( X \), 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 new observation of SCFD \( x^* \) from the centroid is

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

If it is greater than a cutoff distance \( \theta \), we consider that the execution to be malicious. For example, \( B \) in Figure 3 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,

\[
p(x^*) = \frac{1}{\sqrt{||\Sigma||(2\pi)^d}} \exp\left( -\frac{1}{2} dist(x^*, c)^2 \right). \tag{2}
\]

That is, we can choose a \( \theta \) such that the \( p \)-value under the null hypothesis is less than a significant level \( p_0 \), for instance, 1% or 5%. 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.

---

\(^{23}\) The gray-colored points (circle, triangle, rectangle) are system call frequency distributions 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 ellipsoid are legitimate with respect to the cluster.
First, let \( z \) be a Mahalanobis distance from a multivariate normal distribution. Then,

\[
\int_0^\theta c \cdot e^{-\frac{1}{2}z^2} \, dz = 1 - p_0, \tag{3}
\]

where \( c \) is a normalizing constant that satisfies (3) with \( \theta = \infty \) and \( p_0 = 0 \) by the definition of a probability density function. This results in \( c = \frac{1}{1.25331} \) because

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

where \( \text{erf}(z) \) is the error function and is 1 and 0 for \( z = \infty \) and \( z = 0 \), respectively. Accordingly, (3) becomes

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

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

\[
\theta = \frac{\text{erf}^{-1}(1 - p_0)}{0.707107}. \tag{4}
\]

For \( p_0 = 1\% \) and \( 5\% \), \( \theta \approx 2.57583 \) and \( 1.95996 \), respectively. Figure 4 shows the cutoff distance for \( 0\% \leq p_0 \leq 100\% \). The cutoff distance is not bounded (i.e., \( \theta = \infty \)) when \( p_0 = 0\% \) and is 0 when \( p_0 = 100\% \).

### C. Intrusion Detection for Multiple Execution Contexts Using Global k-means Clustering

In general, an application may show widely varying system call distributions due to multiple execution modes and a wide range of possible inputs. In such scenarios, finding a single multivariate normal distribution (i.e., a single cluster/centroid) for the whole set can result in inaccurate models because it would include even many non-legitimate points that belong to none of the execution contexts – i.e., the empty space between clusters in Figure 3. 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 (or centroid) 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 due to a 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 (24) 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 \( c(x^n) \), i.e.,
   \[ c(x^n) = \arg \min_{c_k \in C} \text{dist}(x^n, c_k). \tag{5} \]
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,

\[
\text{total-dist}(X, C) = \sum_{n=1}^N \text{dist}(x^n, c(x^n)), \tag{6}
\]

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 \(^\text{25}\) to find the number of clusters as well as the initial assignments that lead to deterministic accuracy. Algorithm \(^\text{1}\) illustrates the global k-means algorithm. Given a training set \(X\), the maximum number of clusters \(\text{MAX}_K\), and the total distance bound \(\text{Bound}_{\text{DP}}\), the algorithm finds the best number of clusters and assignments. This is an incremental learning algorithm that starts from a single cluster consisting of the entire data set. In the case of \(k = 2\), the algorithm considers each \(x^\ast \in X\) 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 \(^\text{6}\). 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 \(\text{MAX}_K\) or the total distance value becomes less than the total distance bound \(\text{Bound}_{\text{DP}}\). Note that the total distance in \(^\text{6}\) 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 standard k-means algorithm uses the Euclidean distance and thus the centroids of the initial clusters are the data points that were picked first. 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 new observation \(x^\ast\) to each cluster using \(^\text{1}\). The legitimacy test of \(x^\ast\) is then performed by finding the closest cluster, \(c^\ast\), using \(^\text{5}\).

Thus, if
\[
dist(x^\ast, c^\ast) = \min_{c_i \in C} dist(x^\ast, c_i) > \theta
\]
for a given threshold \(\theta\), we determine that the execution does not fall into any of the execution contexts specified by the clusters since \(dist(x^\ast, c_i) > \theta\) for all \(i = 1, \ldots, k\). We then consider the execution to be malicious. As an example, for the new observation \(C\) in Figure 3 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 would have a smaller ellipsoid in the Euclidean space.

D. Dimensionality Reduction

The number of system call types, i.e., \(D\), is quite large in general (e.g., 312 system call types in Linux 3.2 for x64). Thus, the matrix calculations in Equation \(^\text{1}\) might result in an unacceptable amount of analysis overhead\(^\text{6}\). However, a real-time embedded application normally uses a limited subset of system calls. Furthermore, we can significantly reduce the dimensionality by ignoring system call types that never vary. Consider Cluster 4 from Figure 3. 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 presented above, we reduce \(S\) to \(S' = \{s_{d_1}, s_{d_2}, \ldots, s_{d'}\}\), where \(D' \leq D\), such that the variance of \(x_d\) in the entire training set \(X\) is zero for each \(s_d \in S - S'\). Note that \(S - S'\) includes any system call types that never appeared in the training set. However, we should still be able to detect any changes in such system calls. Thus, we merge all the variables in \(S - S'\) by calculating the sum \(\sigma = \sum_{s_d \in S - S'} x_d\), where \(x_d = x_{d_1} = x_{d_2} = \cdots = x_{d'}\). Then, in the monitoring phase, we test if \(\sigma - (\sum_{s_d \in S - S'} x_d)\) is non-zero. If it is, the application shows a variation in \(S - S'\), which should be considered suspicious. In case \(D'\) is still large, one may apply a statistical dimensionality reduction technique such as Principal Component Analysis (PCA) \(^\text{26}\) or Linear Discriminant Analysis (LDA) \(^\text{27}\).

III. ARCHITECTURAL SUPPORT FOR SYSTEM CALL MONITORING

Most of the existing system call-based intrusion detection systems rely on the operating system to provide the information, say, by use of auditing modules \(^\text{20}, \text{28}\). While this provides the ability to monitor extensive properties such as system call arguments, it requires us to assume that the operating system itself remains trustworthy. In this paper we avoid this problem by proposing new architectural modifications. The architecture builds upon the SecureCore architecture \(^i\) tuyoossecurecore2013 that enables a trusted on-chip entity, e.g., a secure core, to continuously monitor the run-time behavior of applications on another, potentially untrusted entity, the monitored core, in a non-intrusive manner. In this section we describe the modifications to SecureCore that enable us to monitor the system calls. We refer interested readers to \(^\text{14}\) for the full details about the SecureCore architecture.

A. Overview

Figure 5 shows the overall architecture for system call monitoring. It consists of \((a)\) a secure core, \((b)\) a monitored...
core, (c) an on-chip system call tracing module (SCTM) and (d) a scratch pad memory (SPM). The secure core monitors the usage of system calls by applications executing on the monitored core. The SCTM extracts relevant information from the monitored core and then writes it to the SPM. A monitoring process on the secure core then uses this information to check whether the run-time behavior has deviated from the expected behavior, i.e., the profile that we obtained from the analysis.

We obtain the profile of ‘normal’ executions in a similar manner: the monitoring process collects SCFDs using the SPM under sanitized conditions. We then apply the learning algorithm explained in Section IV. The resulting ‘normal’ profile (per application) is then stored in a secure memory location.

B. System Call Tracing Module (SCTM)

The system call tracing module (SCTM) is located between the monitored and secure cores as shown in Figure 5. SCTM tracks how many times each application on the monitored core uses each system call type (i.e., SCFDs). The main point is to catch the moment each call is invoked. We are able to do this because, in most processor architectures, a specific instruction is used by x86 and x64 operating system kernels. Hence, the execution of the special instruction has other modes as well: (i) INST_BEGIN and (ii) INST_END that demarcate the region where we are collecting the usage of system calls. Once INST_END completes, the monitor retrieves the data collected during the recently completed region of execution and applies the detection algorithm. The data is reset with the execution of an INST_BEGIN. While an attacker may try to execute malicious code block before BEGIN or after END to avoid detection, we can catch such situations because there should be no system call execution during that point in the code. Thus, in such cases, the SCTM would immediately raise an alarm. Also, an attack may skip any of the special instructions or modify any of the values. Again, these cannot help the attacker hide malicious code execution because the system call distribution would need to be consistent with the profile. Also, watchdog timers can be used to check whether the applications are executing the special instructions in time.

Figure 6 shows how an open system call invocation is detected. As explained above, the system call number, which is 5 in Linux, is written to r0 register. Thus, by looking at the value of the register, we can track which system call is being invoked. When an sc instruction is executed, SCTM takes the PID register and the r0 register values. It then updates the corresponding SCFD entry in the scratch pad.

After the execution of the system call, the secure monitor is able to map each application to corresponding PIDs. The above registration process is carried out by a special instruction, INST_REG_AID, as described in the figure. The special instruction has other modes as well: (i) INST_BEGIN and (ii) INST_END that demarcate the region where we are collecting the usage of system calls. Once INST_END completes, the monitor retrieves the data collected during the recently completed region of execution and applies the detection algorithm. The data is reset with the execution of an INST_BEGIN. While an attacker may try to execute malicious code block before BEGIN or after END to avoid detection, we can catch such situations because there should be no system call execution during that point in the code. Thus, in such cases, the SCTM would immediately raise an alarm. Also, an attack may skip any of the special instructions or modify any of the values. Again, these cannot help the attacker hide malicious code execution because the system call distribution would need to be consistent with the profile. Also, watchdog timers can be used to check whether the applications are executing the special instructions in time.

Figure 6 shows how an open system call invocation is detected. As explained above, the system call number, which is 5 in Linux, is written to r0 register. Thus, by looking at the value of the register, we can track which system call is being invoked. When an sc instruction is executed, SCTM takes the PID register and the r0 register values. It then updates the corresponding SCFD entry in the scratch pad.

As another example, int 80h instruction with eax register is used by x86 and syscall instruction with rax register is used by x64.

As another example, int 80h instruction with eax register is used by x86 and syscall instruction with rax register is used by x64.

As another example, int 80h instruction with eax register is used by x86 and syscall instruction with rax register is used by x64.

As another example, int 80h instruction with eax register is used by x86 and syscall instruction with rax register is used by x64.
memory (see Figure 7). An entry is a contiguous memory region of length $2D + 4$ bytes, where $D$ is the total number of system calls provided by the OS. Using the PID, SCTM locates the corresponding entry and then increments the counter of system call $d$ if the value in $r0$ was $d$. The sizes of the SPM and each entry field are implementation-dependent. In our implementation, we assume at most 382 system call types (which are enough to cover what Linux provides) that results in the size of an entry being 768 bytes at most. Thus, an SPM of size 8KB can provide a space for around 10 applications to be monitored simultaneously. The SPM can be accessed only by the secure core. It is mapped to a range of the secure cores address space that is protected by a hypervisor [14] or a hardware-enforced memory protection mechanism [30]. When an INST_END is executed, the secure monitor reads the corresponding entry from the SPM, finds the profile for the corresponding application using the PID to AID map, then verifies the legitimacy of the execution.

IV. EVALUATION

In this section, we first present the implementation details for our prototype (Sec. IV-A), the application model for our experiments (Sec. IV-B), some attack scenarios that are relevant to this application (Sec. IV-C). We then evaluate the efficiency and the effectiveness of the proposed approach, by comparing it with one of the well-known sequence-based approach (Sec. IV-D). Finally, and And finally we discuss the implementation and possible improvements (Sec. IV-E).

A. System Implementation

We implemented a prototype of the proposed intrusion detection system on Simics [21] as shown in Figure 8. Simics is a full-system simulator that can emulate a hardware platform including real firmware, device drivers and also allows processor architecture modifications. We used the dual-core MPC8641D Processor [29, 31] platform. Each core runs at 1350MHz and the system has a memory of 1GB and runs Linux 2.6.23. The SCTM was implemented by extending the sample-user-decoder on Simics. This allows us to implement the necessary ISA modification as described in Section III-B. The SPM has a total size of 8KB.

B. Target Application Model

Figure 9 (a) shows the target application. Each job instance of the application (period of one second) cycles through the following steps: (i) retrieve a raw image from a camera, (ii) compresses it to a JPEG format, (iii) upload the image file to the base station through FTP and finally (iv) write a log via a HTTP post. This type of application model (image upload → processing → communication) can be found in modern unmanned aerial vehicles (UAVs) that are used for surveillance or environmental studies [32]. Typically these systems have real-time constraints on their application tasks.

The distributions of the system call frequencies exhibited by this application is mainly affected by the stages after the JPEG compression. While the raw image size is always fixed (e.g., 2.6MB for 1280 x 720 resolution), a JPEG image size can vary (27KB – 97KB) because of compression. This results in a variance in the number of read and write system calls. To increase the complexity of the application and also to include further variations in the distribution of system calls, we added an additional code branch before the FTP upload stage that behaves as follows: the system could randomly skip the image upload process (based on a probability of 0.5). This affects the number of occurrences of some network and file-related system calls during actual execution. Hence, the application has two legitimate flows, Flow 1 and Flow 2 as shown in Figure 10, that also shows the system call types used at each stage of the execution flow.

We use this type of application model for the following reasons: (a) Simics, being a full system simulator that executes on a “host” system, is not fast enough to be able to control an actual real-time control system. Hence, we need to develop an application model that it can simulate; (b) we still need to demonstrate, in the simplest possible way, how our SCFD-based intrusion detection system works – this application model is able to highlight the exact mechanisms and even its limitations. Note: In fact, this example is crafted with the intention of showing more variance than many real-time
systems. Hence, if our detection method can catch changes in the system call distributions of this application, then it can very well detect similar issues in RTS that show less variance.

C. Attack Scenarios

We consider the following attack scenarios for the application described above:

1) **Attack 1**: The attack code steals user authentication information used to connect to the base station’s FTP server and sends it to an adversary HTTP server. *This attack invokes the same HTTP logging calls used by the legitimate executions.*

2) **Attack 2**: It uploads the image that was just encoded by the application to an adversary FTP server. *This attack also uses the same functions used by the legitimate FTP upload.*

3) **Attack 3**: It modifies the raw image array received from the camera. The attack erases the array by calling memset from the camera. *The attack erases the array by calling memset.*

4) **Attack 4**: It is a real shellcode targeted for Linux on PowerPC and executes execve to spawn a shell (/bin/sh) [33]. In general, a shellcode can be injected by data sent over a network or from a file and can be executed by exploiting buffer overflow or format string vulnerabilities. In our implementation, the shellcode is stored in char shellcode[] and is executed by __asm__('\'b shellcode''') when enabled.

The attack codes execute at spots marked in Figures [9] and [10] when enabled. Our method is independent of where they happen since SCFDs do not care about the sequences of system calls.

D. Results

To profile the distribution of system calls, we executed the system under normal conditions (i.e., no attack code activation) 2,000 times. The target application used 14 types of system calls (apart from those shown in Figure [10] in the appendix it also used futex, rt_sigreturn and brk system calls. We used the learning algorithm (Section II) with settings, $\text{MAX}_k = 10$ and the total distance bound $\text{Bound}_{d_0}$ of 1,000 on the resulting traces. Since the application has a few control flow paths, it was safe to assume at most 10 clusters. The cutoff distance, $\theta$, that attests to the legitimacy of a new SCFD, is 1.95096; i.e., the significance level is 5%. We also tested for $p_0 = 1\%$, i.e., $\theta = 2.57583$.

Table II summarizes the results of applying our analysis. The first row shows the mean and the standard deviation of the SCFDs in the training set. The algorithm first reduce the dimensionality of the results from 14 to 10 by removing the system call types that show zero variance in the training set. The variations of write and read are due to the JPEG compression and the FTP uploading phases. The latter also affects the network and file-related system call types. The global $k$-means algorithm stopped at $k = 5$ (the moment when the total distance becomes less than the bound $\text{Bound}_{d_0}$) resulting in five clusters as shown in the same table. From these results, especially from observing the mean values of the system call types other than write and read, we can infer that Clusters 2 and 3 are from a similar execution context while the others are from a different context. We also observe that the former group corresponds to Flow 1 because of the additional system calls required for the FTP transfer. Also, the fewer number of write and read system calls of the second group suggest that they belong to Flow 2. As expected, within each group clusters are distinguished by write and read due to the varying sizes of images that are compressed. The clustering results would be similar if $\text{MAX}_k$ was set to, for example, 2. In this case, one cluster would have the points from Clusters 1, 4 and 5 combined but with different centroid and similarly while 2 and 3 would constitute another new cluster. This could, however, blur boundaries between the execution contexts.

Now, we evaluate the accuracy of our intrusion detection methods. We enabled each of the attacks from Section IV-C. For each attack type, we carried out 300 execution instances and measured how many times the monitor detects malicious execution. An execution is considered malicious if any of the following is true: (i) any system call other than the 14 observed types is detected; (ii) any system call whose variance was zero during the profile (4 out of 14 in the case above) actually exhibits variance or (iii) the distance of an observation from its closest cluster is longer than the threshold. Among these, rule (i) was never observed in the cases of Attacks 1-3 because Attacks 1 & 2 re-used the same functions from normal executions and Attack 3 makes no system calls at all. Table III summarizes the results of our detection methods as well as those of the sequence-based approach (explained in Section IV-F). Hence, the results of our detection methods are.
as follows.

1) **Attack 1 (HTTP post):** All the executions were classified as malicious based on rule (ii) above since one additional brk and sendto, each, were invoked. We tested the executions again, after removing such obvious situations. The results, however, did not change – **all the malicious executions were caught, again, by our monitor.** Because of the additional HTTP request by the attack code, socket, connect, close and stat were called more in both Flows 1 and 2. System call types other than the ones mentioned here were consistent with the profile. With $p_0 = 1\%$, the results were the same since the use of the additional system calls already increased the distances from clusters to lie outside the acceptable boundaries.

2) **Attack 2 (FTP upload):** If the attack code is executed on Flow 1, it is easily caught because of the additional FTP transfer. As with the above, this attack changes some network related system calls. This is enough to make the SCFDs fall outside the legitimate regions. The attack reads the image file as well – this increases the usage of read system calls thus further highlighting the anomalous behavior.

On the other hand, if the attack is launched on Flow 2 (that skips the FTP upload to the base station), it may not be as easy to detect. Since the attack uses the same functions that are invoked by legitimate code, it looks like the application is following Flow 1 (where the FTP upload is actually legitimate). In this case, only 1\% of the malicious code executions were caught. The detection rates would be significantly higher if the attacker either used different images that vary in size or used code that utilizes different combinations of system calls. The latter case would also hold for the HTTP post attack. **Note:** while it looks like the detection mechanism was not successful in this case, that is only because we tailored the attack instance to closely match legitimate execution (especially due to our knowledge of the detection methods); many attacks will not be able to match legitimate execution in such a precise manner and will end up being caught.

3) **Attack 3 (Image modification):** This attack does not use any system calls; it just changes the values of the data. However, this may affect executions that follow, especially ones that depend on the data – the JPEG compression in our case. The attack code resets the raw image data by using memset. This is compressed by the JPEG encoder that produces 15 KB of black images. This attack was always caught by our monitor mainly because such image sizes were not typical during normal execution. Hence, calls to read and write were much less frequent as compared to the normal execution (where these calls were used often to write the larger images to files or upload on FTP servers). The attack could have circumvented our detection method if, e.g., the raw image is just replaced with another that has a similar after-compression size as the original or only a part of the image is modified. But performing either of these actions may also trigger the use of additional system calls that would be caught by our monitor.

4) **Attack 4 (Shellcode):** This attack was easily detected since it uses execve which was never observed during the profiling phase. Furthermore, it was followed by a bunch of system calls including open, mmap, access, geteuid, etc. This was due to the execution of a shell, /bin/sh, spawned by the injected execve. In fact, INST_END was not executed since execve does not return on success. Nevertheless, the attack could be detected because a watchdog timer was used to wake up the secure monitor that then checks the application’s SCFD traced until the timer expires. From this experiment it can be expected that our method can detect a more sophisticated shellcode that uses unusual system calls, such as setreuid, setregid, etc.\[9\]

\[9\]If the attacker modified the compressed image, our method cannot detect it because the system call usage would never change. This, however, does not fall into our adversary model explained in Section [1].

\[10\]The shellcode used in our experiment is simpler than the ones targeted for Linux/x86 due to the scarcity of sophisticated shellcode for Linux/PowerPC.
The **false positive rate** is perhaps just as important as the detection rate because frequent false alarms degrade the system availability. To measure the false positive rates, we ran the monitoring system without activating any attacks and measured how many times the secure monitor classifies an execution as being suspicious. Most false positives in these tests were due to the images sizes that, when compressed, fell below the normal ranges. For the cut-off distance \( \theta \) with \( p_0 = 5\% \), 35 out of 2,000 executions (1.75\%) were classified as malicious. With \( p_0 = 1\% \), i.e., a farther cut-off distance, it was reduced to just 17 (0.85\%). 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 \( p_0 = 1\% \). This is something that system designers must consider when implementing our intrusion 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.

These results show that our methods can effectively detect malicious code executions without relying on complex analysis. 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.

### E. Time Complexity

To evaluate the time complexity of the proposed detection method, we measured the number of instructions retired by the function that finds the closest cluster (Eq. 5) among the five clusters given a new observation and the average time to perform the analysis\(^\text{11}\). As Table II shows, the detection process is very quick. 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, since \( (x^* - \mu)^T \Sigma^{-1} (x^* - \mu) \), the first multiplication takes \( O(D^2) \) and the second one takes \( O(D) \). 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 \( O(D^3) \). Note, again, that the monitoring and detection methods are not in the critical path, i.e., they do not affect the execution of the applications we monitor since they are offloaded onto the secure core. 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 II-D and Section IV-E). This deterministic time complexity makes our approach more suitable for real-time embedded systems.

| Type   | SCFD | stide | Note                                               |
|--------|------|-------|---------------------------------------------------|
| Attack 1 | all  | all   | Both: detection due to additional network-related system calls |
| Attack 2 (Flow 1) |   | 57%   | SCFD: detection due to additional network- and file-related system calls stide: detection due to FTP session error, 0% detection for successful attack |
| Attack 2 (Flow 2) | 1%  | 57%   | SCFD: not differentiable from Flow 1 stide: detection due to FTP session error, 0% detection for successful attack |
| Attack 3 | all  | 0% (N=5) all (N=7) | SCFD: too small image size stide: short sequence cannot capture shortened write chain |
| Attack 4 | all  | all   | Both: execve was never seen |

### TABLE III

**Detection Rates of SCFD Method and stide Method.**

| Type   | SCFD | stide | Note                                               |
|--------|------|-------|---------------------------------------------------|
| Attack 1 | all  | all   | Both: detection due to additional network-related system calls |
| Attack 2 (Flow 1) |   | 57%   | SCFD: detection due to additional network- and file-related system calls stide: detection due to FTP session error, 0% detection for successful attack |
| Attack 2 (Flow 2) | 1%  | 57%   | SCFD: not differentiable from Flow 1 stide: detection due to FTP session error, 0% detection for successful attack |
| Attack 3 | all  | 0% (N=5) all (N=7) | SCFD: too small image size stide: short sequence cannot capture shortened write chain |
| Attack 4 | all  | all   | Both: execve was never seen |

### F. Comparison with Sequence-based Approach

To compare our detection methods against known, existing approaches, we use the sequence time-delay embedding (stide) \([15, 16]\) method that is the most well-known technique from literature. It works as follows: from a normal trace, construct a database of unique sequences of fixed length \( N \) (by sliding a window of length \( N \)). Then, for a given sequence of length \( N \) observed at run-time, check if the database contains the sequence; if not, it is considered to be malicious. A more advanced method is to calculate the minimal hamming distance between the given sequence and the ones in the database. The simpler version is used here since the latter incurs a significant overhead due to checking against all of the sequences in the database \([15]\). The first method can be easily implemented by a tree that only requires \( N - 1 \) comparisons.

We tested two configurations: (i) \( N = 5 \) and (ii) \( N = 10 \). There are 131 and 175 unique sequences of length 5 and 10, respectively, that were extracted from the normal trace (of 2000 iterations). An iteration (i.e., one execution) is classified as being malicious if any of the sequences of the iteration is classified as malicious. Table III summarizes the results from the detection techniques.

1. Attack 1: The stide method was able to detect Attack 1 because of this particular sequence: sendto-close-write-write-socket; the extra HTTP activity made

### TABLE II

**Time Complexity of Our Analysis**

| # of system call types | Number of instructions | Avg. (Stdev.) of analysis times |
|------------------------|------------------------|-------------------------------|
| 5                      | 2175                   | 0.914 µs (0.553 µs)           |
| 10                     | 4875                   | 2.624 µs (1.405 µs)           |
| 14                     | 8125                   | 5.231 µs (1.965 µs)           |

\(^{11}\)Simics is not a cycle-accurate simulator. Thus, the times are measured on a real machine with Intel Core i5 1.3GHz dual-core processor. The analysis code is compiled with -O0 option. The statistic is based on 10,000 samples.
sendto-close to be too close to the socket call made by the legitimate HTTP logging. In fact, the method could not detect the attack at all with \( N = 3 \) because the sequences could not relate sendto-close with socket. An attacker could have easily circumvented this method by making a few more extra write calls between them.

2) **Attack 2**: For Attack 2, the stide method classified about 57% executions as malicious (for both \( N = 5 \) and 10), irrespective of the flow. In fact, the malicious sequences were purely due to the FTP server error (too many connections) that caused the malicious FTP session to be disconnected. The corresponding sequence is read-write-write-socket that was never seen in the normal trace. In normal cases, there should be more pairs of read-write between connect and socket. In fact, for cases when the connection was successful, the stide method was not able to detect Attack 2 at all. That is, it could not detect the extra FTP operation.

3) **Attack 3**: The stide method performed poorly for Attack 3 as well since the attack only alters the number of read and write calls. In particular, the size of modified images affects the number of write between mmap and munmap calls made when writing the compressed JPEG image to a file. When an attacker modifies the image, the length of the write chain becomes shorter. For \( N = 5 \), none of the executions were classified as being malicious because the write chain was always longer then 5 and thus no sequence of length 5 was able to capture the shorter chains. In fact, \( N \) had to be at least 7 for the method to detect this attack. Note that, if the image size got larger instead and thus made the write call chain longer than usual, this method could not detect this behavior because the only change would be that there are more sequences of write that have length of \( N \) which are legitimate. Our SCFD-based method, on the other hand, can detect this type of attack.

4) **Attack 4**: Attack 4 was easily caught by the stide method because execute system call was never seen in the normal trace. Notice that our methods can detect such anomalies by carrying out a single analysis on the collected system calls while sequence-based methods should perform an analysis on every call made.

The stide method did not raise any false positives for \( N = 5 \). This is particularly because, as in the case of Attack 3, the method cannot be sensitive to varying image sizes (this was the main cause of false positives by the proposed approach). Only extremely small image sizes could raise an alarm and sequences of length 5 were too short for false alarms. For \( N = 10 \), the false positive rate was around 0.5%. This was mainly due to the images with small after-compression size. In general, the false positive rate of sequence-based methods increase with \( N \) as there are more numbers of distinct patterns.

Lastly, we evaluated the analysis overhead of the stide method with the same setup used in Section \[V.C\]. Note again that in sequence-based methods, an analysis is performed for each system call made. In this evaluation, however, we measured the time required to analyze all the sequences of each execution iteration in a batch so that it can be compared with our SCFD-based method. The stide method took around 260\( \mu s \) and 350\( \mu s \), on average, to analyze one iteration (in batch) for \( N = 5 \) and 10, respectively. Although these numbers do not seem significant, the overhead could be much higher if the system were to indeed invoke the analysis for each system call made. This could be more problematic if the application makes a lot of system calls and multiple applications are being monitored at the same time.

In this evaluation, we tested a simple sequence-based approach. Although better detection accuracy could have been achieved by a complex analysis (such as [18]), most of the sequence-based variants experience similar limitations, viz., that the high-level execution contexts cannot be learned and every single system call should be analyzed and thus the analysis is highly dependent on the complexity of the application.

### G. Limitations and Possible Improvements

One of the limitations of our detection algorithm is that it checks for intrusions after execution is complete (at least for that instance). Thus, if an attack tries to suddenly break the system, we cannot detect or prevent it. However, one can increase the chances of detection such problems by splitting the whole execution range into blocks [14] and checking for the distribution of system calls made in each block as soon as the execution passes each block boundary. This, however, would need more computation in the secure core at run-time, more storage in the SPM and 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 [14]. Since some blocks may use very few system calls (perhaps none) 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 themselves do not show unpredictable timing behavior. This makes it much harder for an attacker who imitates system calls [25] or who replaces certain system calls with malicious functions [22].

### V. RELATED WORK

Forrest et al. [9] build a database of look-ahead pairs of system calls; for each system call type, what is the next \( i^{th} \)
system call for $i = 1, 2,$ up to $N.$ Then, given a longer sequence of length $L > N,$ 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 $N$-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 all the 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 [17] proposes a finite state machine (FSM) based prediction model to relax these requirements and Eskin et al. [18] further improves by employing a wild-card for compact sequence representation. Other prediction models such as Hidden Markov model (HMM) [16] and Markov chains [19] have also been explored. A similar approach to our work is [11], 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. [10] introduce several techniques to test anomalies in argument lengths, character distribution, argument grammar, etc. Maggi et al. [36] use a clustering algorithm to group system calls invocations that have similar arguments.

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., proposed Nitro, a hardware-based system call tracing system where system calls made inside virtual machines in manner similar to ours (Section II-B). We note that our detection method (Section II) 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 [38] and user-level processes for system call interposition [39].

The SecureCore architecture [14] 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. There also exists some work in which a multicore processor (or a coprocessor) is employed as a security measure, such as [13].

VI. CONCLUSION

In this paper we presented a lightweight intrusion detection method that uses application execution contexts learned from system call frequency distributions of real-time embedded applications. We demonstrated that the proposed detection mechanism could effectively detect anomalous behavior due to changes in high-level execution contexts, while the sequence-based approaches could not identify such situations well. We also proposed certain architectural modifications to aid in the monitoring and analysis process. As future work, we plan to combine the SCFD approach with the timing-based approach as described in Section IV-G. Also, we intend to implement the proposed architecture on a soft processor core and to evaluate our method with real-world applications.

REFERENCES

[1] K. Koscher, A. Czeskis, F. Roesner, S. Patel, T. Kohno, S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, and S. Savage, “Experimental security analysis of a modern automobile,” in Proc. of the IEEE Symposium on Security and Privacy, 2010, pp. 447–462.
[2] D. Shepard, J. Bhatti, and T. Humphreys, “Drone hack: Spoofing demonstration on a civilian unmanned aerial vehicle,” GPS World, Aug. 2012.
[3] S. Ravi, A. Raghunathan, P. Kocher, and S. Hattangady, “Security in embedded systems: Design challenges,” ACM Trans. Embed. Comput. Syst., vol. 3, no. 3, pp. 461–491, Aug. 2004.
[4] S. Mohan, “Worst-case execution time analysis of security policies for deeply embedded real-time systems,” SIGBED Rev., vol. 5, no. 1, pp. 1–2, 2008.
[5] M. Handley, V. Paxson, and C. Kreibich, “Network intrusion detection: evasion, traffic normalization, and end-to-end protocol semantics,” in Proc. of the conference on USENIX Security Symposium, 2001.
[6] R. Sommer and V. Paxson, “Outside the closed world: On using machine learning for network intrusion detection,” in Proc. of the IEEE Symposium on Security and Privacy, 2010, pp. 305–316.
[7] M. Abadi, M. Budi, U. Erlingsson, and J. Ligati, “Control-flow integrity principles, implementations, and applications,” ACM Trans. Inf. Syst. Secur., vol. 13, no. 1, pp. 4:1–4:40, Nov. 2009.
[8] J. Criswell, N. Dautenhahn, and V. Adve, “Kcofi: Complete control-flow integrity for commodity operating system kernels,” in Proc. of the IEEE Symposium on Security and Privacy, 2014.
[9] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff, “A sense of self for unix processes,” in Proc. of the IEEE Symposium on Security and Privacy, 1996, pp. 120–128.
[10] D. Mutz, F. Valeur, G. Vigna, and C. Kruegel, “Abnormal system call detection,” ACM Trans. Inf. Syst. Secur., vol. 9, no. 1, pp. 61–93, Feb. 2006.
[11] I. Burguera, U. Zurutuza, and S. Nadji-Tehrani, “Crowdroid: Behavior-based malware detection system for android,” in Proc. of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices, 2011.
[12] C. Zimmer, B. Bhatt, F. Mueller, and S. Mohan, “Time-based intrusion detection in cyber-physical systems,” in Proc. of the ACM/IEEE International Conference on Cyber-Physical Systems, 2010, pp. 109–118.
[13] S. Mohan, S. Bak, E. Betti, H. Yun, L. Sha, and M. Caccamo, “S3A: Secure system simplex architecture for enhanced security and robustness of cyber-physical systems,” in Proc. of the ACM International Conference on High Confidence Networked Systems, 2013.
[14] M.-K. Yoon, S. Mohan, J. Choi, J.-E. Kim, and L. Sha, “SecureCore: A multicore-based intrusion detection architecture for real-time embedded systems,” in IEEE Real-Time Embedded Technology and Applications Symposium, 2013, pp. 21–31.
[15] S. A. Hofmeyr, S. Forrest, and A. Somayaji, “Intrusion detection using sequences of system calls,” J. Comput. Secur., vol. 6, no. 3, pp. 151–180, 1998.
[16] C. Warrender, S. Forrest, and B. Pearlmutter, “Detecting intrusion using system calls: alternative data models,” in Proc. of the IEEE Symposium on Security and Privacy, 1999.
[17] C. Marceau, “Characterizing the behavior of a program using multiple-length n-grams,” in Proc. of the workshop on New security paradigms, 2000.
[18] E. Eskin, “Modeling system calls for intrusion detection with dynamic window sizes,” in Proc. of DARPA Information Survivability Conference and Exposition II, 2001.
[19] S. Jha, K. Tan, and R. A. Maxion, “Markov chains, classifiers, and intrusion detection,” in Proc. of the 14th IEEE workshop on Computer Security Foundations, 2001, pp. 206–219.
[20] G. Parmer and R. Wes, “Hijack: Taking control of cots systems for real-time user-level services,” in Proc. of the IEEE Real-Time and Embedded Technology and Applications Symposium, 2007.
[21] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hillberg, J. Hgberg, F. Larsson, A. Moestedt, and B. Werner, “Simics: A full system simulation platform,” IEEE Computer, vol. 35, no. 2, pp. 50–58, 2002.

[22] “Hijacking system calls with loadable kernel modules,” http://r00tkit.me/?p=46.

[23] P. C. Mahalanobis, “On the generalized distance in statistics,” Proc. of the National Institute of Sciences (Calcutta), vol. 2, pp. 49–55, 1936.

[24] S. Lloyd, “Least squares quantization in PCM,” IEEE Transactions on Information Theory, vol. 28, no. 2, pp. 129–137, Mar. 1982.

[25] I. Jolliffe, Principal Component Analysis. Springer Series in Statistics, 2002.

[26] R. A. Fisher, “The use of multiple measurements in taxonomic problems,” Annals of Eugenics, vol. 7, no. 7, pp. 179–188, 1936.

[27] “SNARE: System iNtrusion Analysis and Reporting Environments,” http://www.intersectalliance.com/

[28] “e600 Core Reference Manual,” http://www.freescale.com/files/32bit/doc/ref_manual/32bCORERM.pdf.

[29] “Linux/PPC shellcode - execve() of /bin/sh,” http://www.shell-storm.org/shellcode/files/shellcode-89.php.

[30] “Linux/x86-64 shellcode - add user with passwd,” http://www.shell-storm.org/shellcode/files/shellcode-801.php

[31] D. Wagner and P. Soto, “Mimicry attacks on host-based intrusion detection systems,” in Proc. of the ACM conference on Computer and communications security, 2002, pp. 255–264.

[32] F. Maggi, M. Matteucci, and S. Zanero, “Detecting intrusions through system call sequence and argument analysis,” IEEE Transactions on Dependable and Secure Computing, vol. 7, no. 4, pp. 381–395, 2010.

[33] J. Pfoh, C. Schneider, and C. Eckert, “Nitro: hardware-based system call tracing for virtual machines,” in Proc. of the International conference on Advances in information and computer security, 2011, pp. 96–112.

[34] D. Wagner and D. Dean, “Intrusion detection via static analysis,” in Proc. of the IEEE Symposium on Security and Privacy, 2001, pp. 156–168.

[35] K. Jain and R. Sekar, “User-level infrastructure for system call interposition: A platform for intrusion detection and confinement,” in Proc. of the IEEE Symposium on Dependable Systems Security Symposium, 1999.

[36] S. Chen, M. Kozuch, T. Strigkos, B. Falsafi, P. B. Gibbons, T. C. Mowry, V. Ramachandran, O. Ruwase, M. Ryan, and E. Vlachos, “Flexible hardware acceleration for instruction-grain program monitoring,” in Proc. of the International Symposium on Computer Architecture, 2008, pp. 377–388.

[37] D. Y. Deng, D. Lo, G. Mahysa, S. Schneider, and G. E. Suh, “Flexible and efficient instruction-grained run-time monitoring using on-chip reconfigurable fabric,” in Proc. of the IEEE/ACM International Symposium on Microarchitecture, 2010, pp. 137–148.

[38] D. Lo, M. Ismail, T. Chen, and G. E. Suh, “Slack-aware opportunistic monitoring for real-time systems,” in Proc. of the IEEE Real-Time and Embedded Technology and Applications Symposium, 2014.

[39] “MicroBlaze Soft Processor Core,” http://www.xilinx.com/tools/microblaze

[40] S. Chen, M. Kozuch, T. Strigkos, B. Falsafi, P. B. Gibbons, T. C. Mowry, V. Ramachandran, O. Ruwase, M. Ryan, and E. Vlachos, “Flexible hardware acceleration for instruction-grain program monitoring,” in Proc. of the International Symposium on Computer Architecture, 2008, pp. 377–388.

[41] H. Kannan, M. Dalton, and C. Kozyrakis, “Decoupling dynamic information flow tracking with a dedicated coprocessor,” in Proc. of the IEEE/ACM International Conference on Dependable Systems and Networks, 2009, pp. 105–114.

[42] D. Y. Deng, D. Lo, G. Mahysa, S. Schneider, and G. E. Suh, “Flexible and efficient instruction-grained run-time monitoring using on-chip reconfigurable fabric,” in Proc. of the IEEE/ACM International Symposium on Microarchitecture, 2010, pp. 137–148.

[43] D. Lo, M. Ismail, T. Chen, and G. E. Suh, “Slack-aware opportunistic monitoring for real-time systems,” in Proc. of the IEEE Real-Time and Embedded Technology and Applications Symposium, 2014.

[44] “MicroBlaze Soft Processor Core,” http://www.xilinx.com/tools/microblaze