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

Man-Ki Yoon
University of Illinois at Urbana-Champaign
mkyoon@illinois.edu

Sibin Mohan
University of Illinois at Urbana-Champaign
sibin@illinois.edu

Jaesik Choi
Ulsan National Institute of Science and Technology
jaesik@unist.ac.kr

Mihai Christodorescu
Qualcomm Research Silicon Valley
mihai@qti.qualcomm.com

Lui Sha
University of Illinois at Urbana-Champaign
lrs@illinois.edu

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.

CCS CONCEPTS

•Computer systems organization →Embedded systems; •Security and privacy →Intrusion detection systems;

KEYWORDS

Embedded Systems, Security, Anomaly Detection

ACM Reference format:
Man-Ki Yoon, Sibin Mohan, Jaesik Choi, Mihai Christodorescu, and Lui Sha. 2017. Learning Execution Contexts from System Call Distribution for Anomaly Detection in Smart Embedded System. In *Proceedings of the 2nd ACM/IEEE International Conference on Internet-of-Things Design and Implementation, Pittsburgh, PA USA, April 2017 (IoTDI 2017),* 6 pages.
DOI: http://dx.doi.org/10.1145/3054977.3054999

1 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 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 [3–5], 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. Traditional behavior-based anomaly detection systems rely on specific *signals* such as network traffic [15], control flow [6], system calls [7, 9], etc. The use of system calls, especially in the form of sequences [8, 9, 13, 16], 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.

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 [14, 19–21] 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 6 in Section 4.4), 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 and 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 [2] and show that minor modifications to the operating system and architectural supports from modern embedded processors enable us to

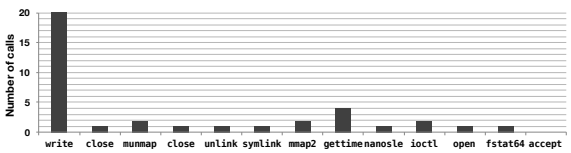


Figure 1: A system call frequency distribution (SCFD).

This work is supported in part by grants from NSF CNS 13-02563, 14-23334, 15-45002, and Navy N00014-14-1-0717. Jaesik Choi is supported by ITRC (Information Technology Research Center) support program (IITP-2016-R2720-16-0007) funded by MSIP, Korea and the Industrial Convergence Core Technology Development Program (No. 10063172) funded by MOTIE, Korea. Any opinions, findings, and conclusions or recommendations expressed here are those of the authors and do not necessarily reflect the views of sponsors.

IoTDI 2017, Pittsburgh, PA USA

DOI: http://dx.doi.org/10.1145/3054977.3054999

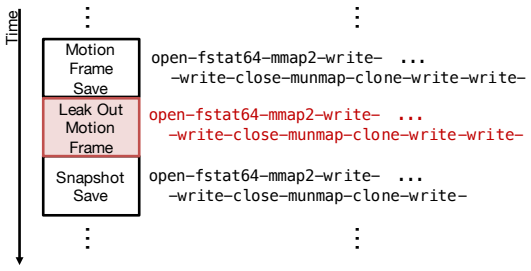


Figure 2: Sequence of system calls made by `Motion` (used in the evaluation in Section 4). An attacker can use the exact same routine used by the legitimate code to leak images out.

monitor and analyze the run-time system call usage of applications in a non-intrusive manner. We use a real-world open-source application [1] and demonstrate that SCFDs can effectively detect certain types of abnormal execution contexts that are difficult for traditional sequence-based approaches.

2 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` [1], 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. 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 since the required pattern lengths are 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. 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. 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 affect executions afterward so that the future system call usage 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. We especially consider stealthy, indirect attacks, e.g., ones that collect important system information or leak out sensitive data while the system/application is functioning normally; or attacks that degrade the availability of such systems. 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 and such attempts can be detected by other techniques [18–20]. Also, we do not make any assumptions as to how the compromised program is present on the device. The attacker may have installed the modified target program in the system or induced users to download the modified source code or the executable binary using, for example, a social engineering tactic.

Assumptions: (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. 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. (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 [17–19]).

3 ANOMALY DETECTION USING SYSTEM CALL FREQUENCY DISTRIBUTIONS

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 σ^n of $\mathcal{S}$. Let us denote the n^{th} *system call frequency distribution* as $\mathbf{x}^n = [m(\sigma^n, s_1), m(\sigma^n, s_2), \ldots, m(\sigma^n, s_D)]^T$, where $m(\sigma^n, s_d)$ is the multiplicity of the system call of type d in σ^n. Hereafter, we simplify $m(\sigma^n, s_d)$ as x_d^n. Thus, $\mathbf{x}^n = [x_1^n, x_2^n, \ldots, x_D^n]^T$.

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 $\mathbf{X} = [\mathbf{x}^1, \mathbf{x}^2, \ldots, \mathbf{x}^N]^T$. The

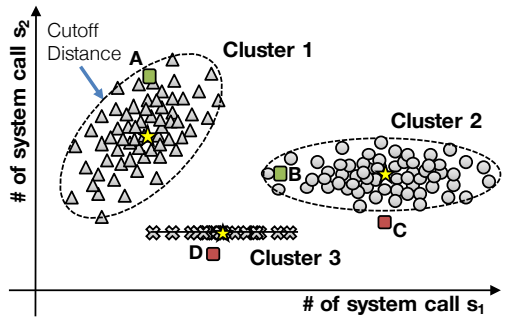


Figure 3: System call frequency distributions for $\mathcal{S} = \{s_1, s_2\}$ and clusters. The gray-colored objects are SCFDs in the training set. Each star-shaped point is the centroid of each cluster. The ellipsoid around each cluster draws its cutoff line.

clustering algorithm (Section 3.2) then maps each $\mathbf{x}^n \in \mathbb{N}^D$ to a cluster $c_i \in C = \{c_1, c_2, \ldots, c_k\}$. We denote by $c : \{\mathbf{x}^1, \cdots, \mathbf{x}^N\} \rightarrow C$ the cluster that $\mathbf{x}^n \in \mathbf{X}$ belongs to.

3.1 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 and structure in such system tends to be fairly limited in what it can do.

For a multivariate distribution, the mean vector $\mu = [\mu_1, \mu_2, \ldots, \mu_D]^T$, where $\mu_d = (\sum_n^N x_d^n)/N$, can be used as the centroid. Figure 3 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 (or L^2-norm) between the new observation $\mathbf{x}^*$ and the mean vector of a cluster. Although the Euclidean distance 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^n and x_2^n 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 [12], defined as $\sqrt{(\mathbf{x}^n - \mu)^T \Sigma^{-1} (\mathbf{x}^n - \mu)}$, for a group of data set $\mathbf{X}$, where Σ is the covariance matrix of $\mathbf{X}$. Notice that the existence of Σ^{-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 changes in such system calls. 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 is in terms of the Euclidean distance, it is more reasonable to determine that C is an outlier and B is legitimate 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 s_2 (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 3 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, $\mathbf{X} = [\mathbf{x}^1, \mathbf{x}^2, \ldots, \mathbf{x}^N]^T$, we calculate the mean vector, μ, and the covariance matrix, Σ, for this data set. It then can be represented as a single cluster, c, whose centroid is defined as (μ, Σ). Now, the Mahalanobis distance of a newly observed SCFD, $\mathbf{x}^*$, from the centroid is

$$dist(\mathbf{x}^*, c) = \sqrt{(\mathbf{x}^* - \mu)^T \Sigma^{-1} (\mathbf{x}^* - \mu)}. \quad (1)$$

If this distance is greater than a cutoff distance θ, we consider that the execution to be malicious. One analytic way to derive this threshold, θ, is to think of the Mahalanobis distance w.r.t. the multinomial normal distribution,

$$p(\mathbf{x}^*) = \sqrt{|\Sigma|(2\pi)^D}^{-1} \exp\left(-\frac{1}{2} dist(\mathbf{x}^*, c)^2\right). \quad (2)$$

That is, we can choose a θ such that the p-value under the null hypothesis is less than a significant level p_0, e.g., 1% or 5%.

3.2 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, 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 $\mathbf{x}^*$ is reduced to identifying the *most probable* cluster that may have generated $\mathbf{x}^*$. If there is no strong evidence that $\mathbf{x}^*$ is a result of an execution corresponding to any cluster then we determine that $\mathbf{x}^*$ is most likely due to malicious execution.

Suppose we collect a training set $\mathbf{X} = [\mathbf{x}^1, \mathbf{x}^2, \ldots, \mathbf{x}^N]^T$ where $\mathbf{x}^n \in \mathbb{N}^D$. To learn the distinct distributions, we use the k-means algorithm [11] 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 $\mathbf{X}$.
(2) Assignment: For each $\mathbf{x}^n \in \mathbf{X}$, assign it to the closest cluster, i.e.,
$$c(\mathbf{x}^n) = \arg\min_{c_k \in C} dist(\mathbf{x}^n, c_k). \quad (3)$$
(3) Update: Re-compute the centroid (i.e., μ and Σ) of each cluster based on the new assignments.

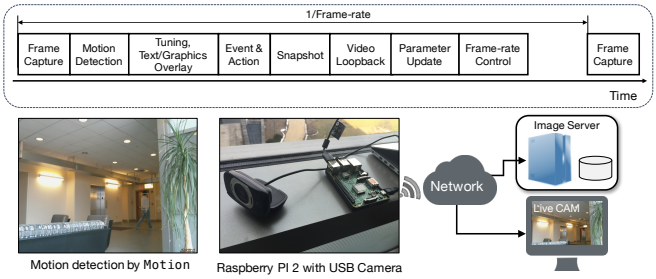


Figure 4: Motion's main execution process (top). The main loop repeats at the frame rate (e.g., 3 frames per second).

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 $\mathbf{x}^n$ to its cluster,

$$total\text{-}dist(\mathbf{X}, C) = \sum_{n=1}^{N} dist(\mathbf{x}^n, c(\mathbf{x}^n)), \quad (4)$$

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 [10] to find the number of clusters as well as the initial assignments that lead to *deterministic accuracy*. Given a training set $\mathbf{X}$ 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 and repeats until either k reaches a pre-defined $\mathsf{MAX_K}$, the maximum number of clusters, or the total distance value becomes less than the total distance bound $\mathsf{Bound_{TD}}$. Note that the total distance in Eq. (4) decreases monotonically with the number of clusters.

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, (μ_i, Σ_i). The legitimacy test of a newly observed SCFD $\mathbf{x}^*$ is then performed by finding the closest cluster, c^*, using Eq. (3). Thus, if $dist(\mathbf{x}^*, c^*) = \min_{c_i \in C} dist(\mathbf{x}^*, c_i) > \theta$ for a given threshold θ, we determine that the execution does not fall into any of the execution contexts specified by the clusters since $dist(\mathbf{x}^*, c_i) > \theta$ 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 cluster. C is malicious since it is outside Cluster 2's cutoff distance.

4 EVALUATION

4.1 Target Application

We use `Motion` [1], an open-source program that monitors camera images and detects motion by tracking changes between image frames as illustrated in Figure 4. It is used for surveillance purpose and provides live streaming and external program execution when certain events (e.g., motion detection, on file creation) are detected.

Figure 4 also show `Motion`'s execution process. The main loop consists of a series of blocks. Each loop starts by capturing an image frame from the camera. When a motion is detected, each frame is saved to the filesystem. Following this, some pre-defined event actions could trigger external programs. 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 when a motion is detected, and the `wput` Linux command is executed to upload the newly created images to a remote server. These external commands are executed by separate processes forked by the main process.

4.2 System Implementation

We implemented a prototype of our SCFD-based anomaly detection system on a Raspberry PI 2 Model B board [2]. 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. All applications and the OS run on Cores 0, 1, 2. 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* available on the Broadcom BCM2836 SoC (System-on-Chip). The secure monitor performs the detection process presented in Section 3 using the SCFD built from the reported information. Since we collect system call usage information at the operating system layer (i.e., software interrupt handler), the OS is our trusted computing base.

4.3 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. For example, Attack 1 can be implemented by inserting the following small piece of code:

```
const char* org_filepath = cnt->conf.filepath;
cnt->conf.filepath = "/path/to/attackers_desired_location";
event(cnt, EVENT_IMAGE_DETECTED, cnt->imgs.image_ring[cnt->
      imgs.image_ring_out].image, NULL, NULL, &cnt->imgs.ima
      ge_ring[cnt->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 and it in turn calls the `put_picture` library routine which saves the current frame image at a desired location. 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, as depicted. Now, the attacker can place this piece of code (followed by a bogus `write` call) between the two file write operations.

4.4 Evaluation Results

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. Figure 5 summarizes the system call sequences made in each loop in normal situations. 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*

```
Motion loop {
    gettimeofday-gettimeofday
    (ioctl)-rt_sigprocmask-ioctl-ioctl-rt_sigprocmask  /* Frame Capture */
    If (motion_detected) {  /* Run external command upon 'on_motion_detected' event*/
        clone
        open-fstat64-mmap2-write- ... -write-close-munmap-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-munmap-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 */
}
```

Figure 5: The system call sequences made by `Motion`.

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

SCFD Training: With the training set obtained from the system under normal conditions, we applied the learning algorithm presented in Section 3. Out of 15 types of system calls used by `Motion`, two types, `select` and `rt_sigprocmask`, showed zero variance. Hence, the algorithm first reduces the dimensionality of SCFDs to 13.

Figure 6 visualizes the training result obtained with settings $\mathtt{MAX_K} = 20$ and $\mathtt{Bound_{TD}} = 1000$. 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 these, we find the following execution contexts:

(1) Cluster 1 represents the case when no event occurs during a loop shown in Figure 5. The loop only takes the current image frame and none of the `if` blocks in Figure 5 execute.
(2) 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 variations in the last `if` block (i.e., webcam remote view-related) in Figure 5.
(3) 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 5). 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.
(4) Cluster 9 corresponds to the case when both motion frame and snapshot files are saved (because of the reasons explained above). It 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.
(5) 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 5). Such differences caused them to stand out in comparison to other clusters. Each one was also not representative enough to create its own cluster.

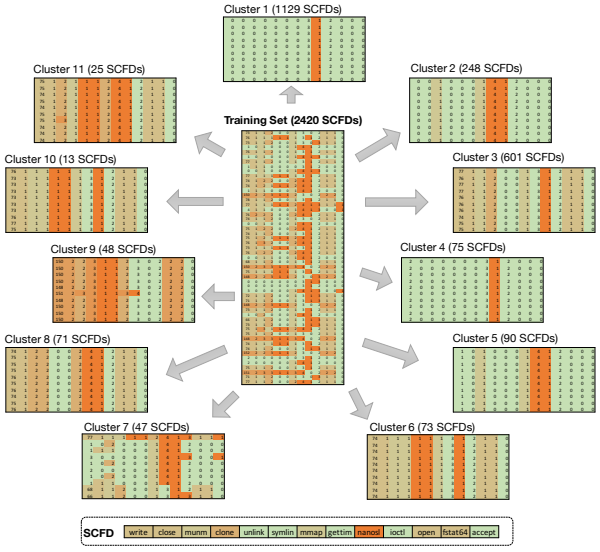


Figure 6: The result of clustering 2420 SCFDs. Each row represents an SCFD of 13 system call types. The training set shows only the unique SCFDs and each cluster shows only 10 SCFD examples that belong to it.

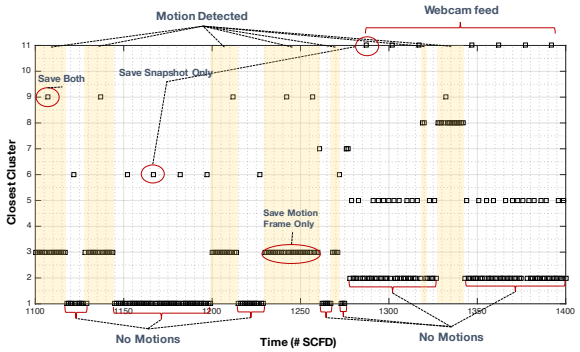


Figure 7: SCFDs in normal situation, their closest clusters assigned by our detection algorithm and the corresponding execution contexts.

Figure 7 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.

Accuracy: Now, we evaluate the *accuracy* of our anomaly detection methods. We enabled each of the attacks from Section 4.3.

(1) Attack 1: We inserted the code block that leaks out the current frame image to the attacker's desired location (as explained in Section 4.3) 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 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 8. The first group (upper-right) looks very similar to the ones in Cluster 9 (in Figure 6) that saves both

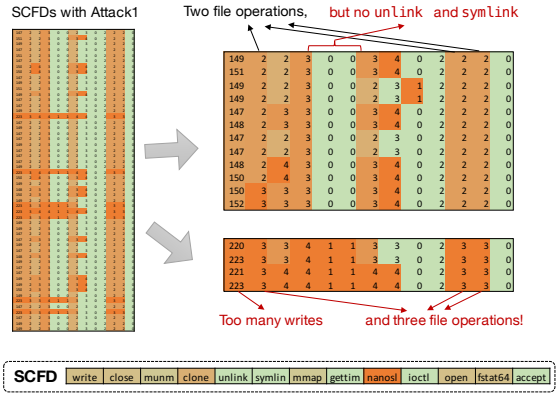
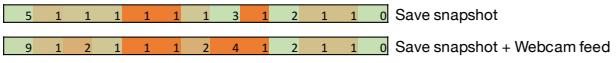


Figure 8: The SCFDs when the attacker leaks out current motion frames (Attack 1). Not all SCFDs are shown.

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 5. 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 into our threat model.

The second group (at bottom-right in Figure 8) consists of SCFDs observed when the inserted code executes between the two legitimate file operations. The resulting SCFDs are 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). This attack produces 14KB of frame images, which results in much less frequent `write` calls.



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, for example, the frame image is just replaced with another that has a similar size as the unmodified ones. However, again, such case is out of scope of our threat model because the system call usage does not change.

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 θ 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.

As explained before, sequence-based approaches may fail to detect abnormal deviations in situations that naturally have a high-level variance in the execution contexts. 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 accuracy of the system call-based anomaly detection for embedded systems.

5 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 method using the topic modeling approach to deal with large-scale heterogeneous behaviors of complex embedded applications.

REFERENCES

[1] Motion. http://www.lavrsen.dk/foswiki/bin/view/Motion/WebHome.
[2] Raspberry PI 2 Model B. https://www.raspberrypi.org/products/raspberry-pi-2-model-b/.
[3] A hacker developed Maldrone, the first malware for drones. *Security Affairs* (Jan 2015). http://securityaffairs.co/wordpress/32767/hacking/maldrone-malware-for-drones.html.
[4] Jeep Hacking 101. (Aug 2015). http://spectrum.ieee.org/cars-that-think/transportation/systems/jeep-hacking-101.
[5] Hackers broadcast live footage from hacked webcams on YouTube and trolls are loving it. (Apr 2016). https://blog.kaspersky.com/2ch-webcam-hack/11961/.
[6] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2009. Control-flow integrity principles, implementations, and applications. *ACM Trans. Inf. Syst. Secur.* 13, 1, Article 4 (Nov. 2009), 4:1–4:40 pages.
[7] Iker Burguera, Urko Zurutuza, and Simin Nadjm-Tehrani. 2011. Crowdroid: Behavior-based Malware Detection System for Android. In *the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices.*
[8] Eleazar Eskin. 2001. Modeling system calls for intrusion detection with dynamic window sizes. In *DARPA Information Survivabilty Conference and Exposition II.*
[9] Stephanie Forrest, Steven A. Hofmeyr, Anil Somayaji, and Thomas A. Longstaff. 1996. A Sense of Self for Unix Processes. In *the IEEE Symposium on Security and Privacy.*
[10] Aristidis Likas, Nikos Vlassis, and Jakob J. Verbeek. 2003. The global k-means clustering algorithm. *Pattern Recognition* 36, 2 (2003), 451 – 461.
[11] S. Lloyd. 1982. Least squares quantization in PCM. *IEEE Transactions on Information Theory* 28, 2 (1982), 129–137.
[12] Prasanta Chandra Mahalanobis. 1936. On the generalized distance in statistics. *the National Institute of Sciences* 2 (1936), 49–55.
[13] Carla Marceau. 2000. Characterizing the behavior of a program using multiple-length N-grams. In *the workshop on New security paradigms.*
[14] Sibin Mohan, Stanley Bak, Emiliano Betti, Heechul Yun, Lui Sha, and Marco Caccamo. 2013. S3A: Secure System Simplex Architecture for Enhanced Security and Robustness of Cyber-Physical Systems. In *the ACM International Conference on High Confidence Networked Systems.*
[15] Robin Sommer and Vern Paxson. 2010. Outside the Closed World: On Using Machine Learning For Network Intrusion Detection. In *the IEEE Symposium on Security and Privacy.*
[16] Christina Warrender, Stephanie Forrest, and Barak Pearlmutter. 1999. Detecting intrusion using system calls: alternative data models. In *the IEEE Symposium on Security and Privacy.*
[17] Peter Wilson, Alexandre Frey, Tom Mihm, Danny Kershaw, and Tiago Alves. 2007. Implementing Embedded Security on Dual-Virtual-CPU Systems. *IEEE Des. Test* 24, 6 (Nov. 2007), 582–591.
[18] Man-Ki Yoon, Mihai Christodorescu, Lui Sha, and Sibin Mohan. 2016. The DragonBeam Framework: Hardware-Protected Security Modules for In-Place Intrusion Detection. In *the ACM International Systems and Storage Conference.*
[19] Man-Ki Yoon, Sibin Mohan, Jaesik Choi, Jung-Eun Kim, and Lui Sha. 2013. SecureCore: A Multicore-based Intrusion Detection Architecture for Real-Time Embedded Systems. In *the IEEE Real-Time Embedded Technology and Applications Symposium.*
[20] Man-Ki Yoon, Sibin Mohan, Jaesik Choi, and Lui Sha. 2015. Memory Heat Map: Anomaly Detection in Real-Time Embedded Systems Using Memory Behavior. In *the ACM/EDAC/IEEE Design Automation Conference.*
[21] Mohammad Mehdi Zeinali Zadeh, Mahmoud Salem, Neeraj Kumar, Greta Cutulenco, and Sebastian Fischmeister. 2014. SiPTA: Signal Processing for Trace-based Anomaly Detection. In *the International Conference on Embedded Software.*