International Journal of Engineering Trends and Technology
ISSN: 2231–5381 / https://doi.org/10.14445/22315381/IJETT-V73I11P120

Volume 73 Issue 11, 280-294, November 2025

Original Article

Performance Comparison for Anomaly Detection Using System Level Traces on Dynamic Utilities

Goverdhan Reddy Jidiga[1], Rambabu Bandi[2], Malla Reddy Adudhodla[3]

[1]*Government Polytechnic, Mahabubnagar, Department of Technical Education, Hyderabad, Telangana, India.*
[2]*Department of CSE, CVR College of Engineering, Ibrahimpattan, RR Dist, Telangana, India.*
[3]*Department of IT, CVR College of Engineering, Ibrahimpattan, RR Dist, Telangana, India.*

[1]*Corresponding Author : jgreddymtech@gmail.com*

Received: 05 April 2025 Revised: 07 November 2025 Accepted: 15 November 2025 Published: 25 November 2025

Abstract - *The adaptive information security combines a wide range of system security approaches and network security methods to create a robust defense strategy. This approach integrates various system models to protect delicate, secretive, and unrestricted information from unlawful admission, misappropriation, alteration, disclosure, interference, and devastation. Anomaly detection is a focused process that investigates the system's data while applications are running. This one suggests utilizing open-source Linux log data for tracing, aimed at enhancing system performance. This innovative method leverages tracing techniques available on the Linux environment, virtually drawing attention to promote performance in live mode. The Key tools like BACKTRACE (bt), LTRACE, PTRACE, and STRACE enable tracing vital system data, including introspection of function calls, investigation of library calls, signals, and a massive quantity of system calls from the stack memory for effective anomaly detection. It is provided that an advocate for the application of adaptive anomaly detection techniques at the data level, particularly through command-level tracing with modern tracing tools. The use of STRACE with LTTng gives better results, and performance is reached beyond threshold levels due to the speed of LTTng (Linux Trace Toolkit Next Generation) compared to other tracing possibilities on system utilities. The overall DR is marked as 99% in all combinations with low FPR compared to individual process tracing tools, and also disclosed the ratio of performance stability about system profiling with dynamic (for live user space) vs. static probes.*

Keywords - *Anomaly Detection, Stack, System Call, Strace, LTTng, Relative Difference, Return Address.*

1. Introduction

The present introspection of anomalies while anomaly detection is a specialized intrusive aspect of intrusion detection and may be described as the progression of categorizing patterns within certain data that deviate from what is recognized as normal behavior [4]. Basically, the anomaly reflects in intrusion in terms of scope, method of detection, and nature of deviation for novel behavior. Essentially, it involves using techniques to determine whether there are security violations in programs by establishing a standard model that reflects normal user or system behavior. It subsequently looks for actions that significantly diverge from these established norms. For instance, consider a scenario where a low-income farmer receives unusually high electricity bills or unexpected tax notices, and this would be an anomaly. In everyday life, recognizing such anomalies in a person's behavior can be quite challenging, as it requires long-term observation of their usual patterns to identify any unusual actions. This raises the question of how it can effectively detect these anomalies in real-time applications, highlighting the importance of systematic learning and practical implementation in this field. Anomaly detection in the live introspection at the system level is a crucial aspect of detecting behavioral actions due to the latest malicious penetrations into the applications, which generate huge log profiles. As per virtual introspection of running processes on the system evades anomaly in this article is actually defined as identifying patterns in specific datasets that deviate from established normal behavior. This approach allows us to determine whether security violations are present within programs by creating a standard model of typical behavior for users or systems and detecting deviations from this model. Identifying anomalies in everyday human behavior is challenging, as it requires ongoing observation over time to discern unusual patterns. Therefore, detecting such anomalies in real-time applications necessitates a methodical approach to learning and analysis.

1.1. Concrete Anomaly Detection

The elementary anomalies are associated with the inventive data activities and are constructed on their incidences or process material; they can be categorized into

three types of abnormal patterns (anomalies) as illustrated in Figure 1.

1. Individual-based anomalies (point-based) are shown in A and C of Figure 1.
2. Contextual anomalies established on substance matter, represented in B of Figure 1.
3. Group abnormalities driven by distinct incidences or uninterrupted data (combined-based) as seen in left in A of Figure 1.

Point anomalies refer to individual instances that do not conform to the expected pattern, while collective anomalies involve larger groups of data that deviate from the norm. In some instances, particularly those related to specific subjects like time series and real-time control, highlight anomalies within their context [12, 13]. All types are depicted in Figure 1. Traditional algorithms used to identify these abnormalities have been notably ineffective, often leading to a high False Positive Rate (FPR) found in [2, 3].

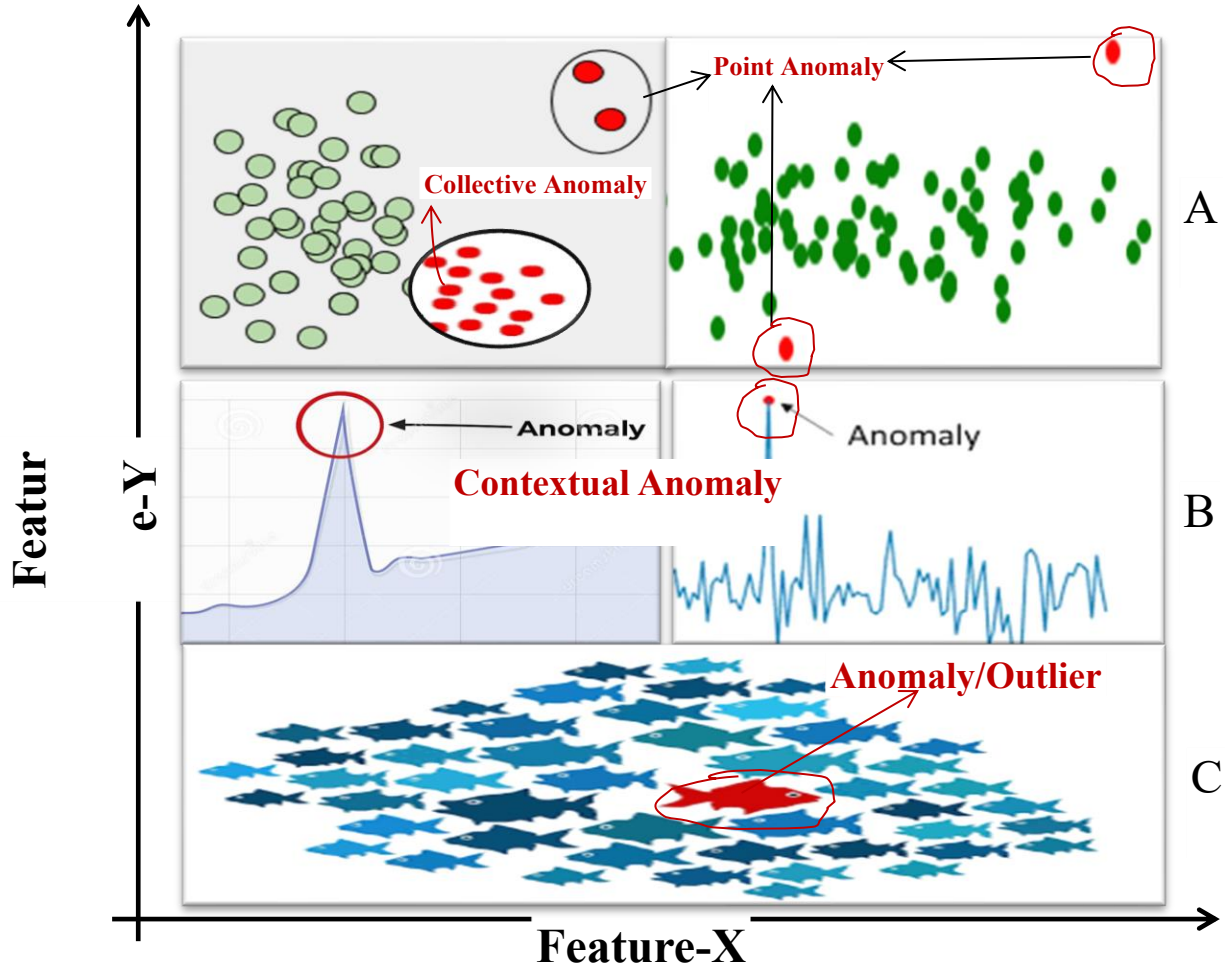

Fig. 1 Interpretation of present anomalies [12, 13]

1.2. Problem Statement and Contribution

An Anomaly Detection System (ADS) is not impervious to genuine threats due to the constantly evolving structure of anomalies and the possible injection of malicious code into applications. Such attacks can potentially cause significant harm to critical infrastructure applications. To effectively address and mitigate anomalies at various data levels in relation to the context of application, it has been witnessed that utilizing standard tracing techniques of Linux alongside enhanced LTTng can utilize correlated techniques to present a more effective approach to anomaly detection, enhancing the classification of anomalies according to the utility's needs. Hence, in this article, the main contribution is an innovative approach by merging the forensic introspection facilities extracted from the virtual running process with the help of strace and LTTng instruments [14, 16, 17]. This work recommends two schemes, along with a comparison that includes a hybrid approach. The proposed schemes, STRACE Line and LTTng Lines, are elaborated on in the following sections. This contribution will fulfill the outcome goals by using lightweight tracing and gradually enhance the performance by the proposed sequence, implemented as data collection, extraction of required data, and detection of anomalies, done after various filtering techniques.

2. Related Works

In the earlier work, the scope is limited to statistical approaches, but indefinite causes of anomaly root labels degrade its performance due to the dynamic nature and the complexity; hence, in this work, the process to enhance the maximum performance potentially while the system is running is considered. This content is elaborated on in a discussion on prior research focused on anomaly detection, utilizing both tracing techniques at the command level and introspection of application levels, presented with clear benefits and drawbacks. This approach primarily employs various tracing techniques available on the Linux platform, including BACKTRACE, PTRACE, STRACE, and LTRACE. These emerging methods are instrumental in extracting vital system data from stack memory for anomaly detection. They prove beneficial in scenarios such as coding exploits at the programming level, debugging running applications, and conducting program tests. Within the realm of scope in detection of anomalies, various benchmark datasets used by people working in this kind of work contain introspected data from system context, like function calls from user space, also system calls and library calls from the entire memory space, mostly prefer live source datasets. The latter significantly enhances online anomaly detection and is particularly valuable for debugging and testing applications.

A concise review of the detection of anomalies using various developed tracing techniques at the level of basic data of the system is given here first. In this method, the detection process utilizes data sourced from the stack. Analysts may apply tracing tricks or employ other tools, which fall into various categories: Stack-based [1-3], Function call-based [1, 8], System call-based [5, 6], PC-based (return address) [1], Library call-based, and additional methods. Some analysts have also suggested leveraging machine learning approaches to identify anomalies at the system data level, utilizing similar data sets previously established in benchmark sets.

Many analysts have developed several methods for detecting anomalies in system data, often focusing predominantly on system call data. With tracing information viewed as an additional layer, it combines system calls and function calls into a sequential arrangement that can assist as a basic set for training to identify unusual activity. In real-time system environments, the high volume of generated system calls can lead to overhead during training and testing exercises. Therefore, it is essential to control data based on necessity while monitoring applications and coding to observe any anomalous behavior.

Some exclusive uses of system calls have encountered failures in specific situations, particularly when debugging and testing real-time applications. This occurs because all system calls and their addresses are flagged as anomalies in online monitoring. To address this issue, a novel approach is necessary, which involves analyzing the relative differences between return addresses. This method proves beneficial for applications that need to load new memory addresses efficiently. The control transfer technique is implemented to find different malicious attacks, but it is found that excessive use of resources delays performance [1]. The limited context of using system call monitoring using traditional modeling can cause a long training period and a drastic occurrence of impossible paths [2, 3]. The modeling nature of system calls with a novel HMM, window-based, N-gram approach is followed by suffering from high FPR, and also a negligible preference [5, 6]. Almost for the first time, strace was utilized to enhance the detection of anomalies working with the UNM dataset, but lacked in finding the necessity of training time, which affects performance due to missing FPR chains [7]. On the other hand, working with log files in Windows tracing by strace and PIN tools slows down the process of function calls, which can vary with the Linux strace utility [8]. The traditional introspection of system calls and static participation of sequence enumerated with hamming distance is also getting attention of peak FPR and convergence time loads hanging on searching lookup tables [11].

In some contexts, working on the LL-MIT dataset may attract more attention to the detection of anomaly, but impossible paths exploits cause, if defined more mimicking attacks, also alarm unauthorized control paths indicate a round of 100% FPR and succeed in DR improvement [9]. The use of UNM PS context also works well in the detection of anomalies, in case of no mis-classification, in this FPR can be reduced well, but finding sequence paths or control paths can detect alarms 100% in mimicry attacks if found lazy classification [10]. New artificial investigation of system calls tracking introduced for both bagging cases with the help of trace compass and LTTng extracted tremendous outcomes, leading to a performance-based detection of anomalies by means of different machine learning types [14].

The LTTng was incepted from dual trace options with deployment of various random and dynamic mechanisms applied to the process to extract the huge data, and also analyzes the real-time performance [16, 17]. The custom trace models enveloped for various cross platforms to extract the potential patterns from the stored database using a generic method [18]. Various real kinds of applications such as server logs, network logs and web logs after extraction of process data dynamically and also consider critical paths to exhibits the performance [20, 21] and also embedded real time application on distributive architecture environment without LTTng to enhance the performance using master-slave synchronization process may time consuming and trace points to be insert, but performance is better as overall [19]. The proposed method can follow the dual collection of log data and extract the unique logs and sequence paths to gain the performance with low time complexity compared to [20]. The dual modes of methods (non-intrusive and intrusive) implemented through live introspection of system calls in

virtual mode can ensure full stack results, and only the overhead indicated in the intrusive method may be due to the applications [15].

3. Materials of Proposed Anomaly Detection

This segment contributes to the STRACE methodology. The recommended effort for abnormality recognition and its accompanying prototypical is illustrated in Figure 2 and elaborated upon in the structure of Figures 4 and 5. It encompasses two key junctures: the determined size of the Training phase, fixed before classification, and the decision-making step of identifying kind anomalies during the Testing phase. All through the Training phase, the steps include extracting control data from the system, requisite data pre-processing is carried out on generated data, then fabricating the datasets which list the Return Address (RAi) and generate sequences between Function Calls (FCSi), and packing these in defined tables. The classification phase ensures the involvement of the testing phase, repeating the first dualistic training steps, generating the novel data sets (new Return Address RAj and new FCSj with updated copy) from attacker exploits using any payloads, and then comparing both old and novel tables to identify anomalies using a defined threshold. System calls are gathered using various tracing techniques in Linux. The total number of system calls and their corresponding program counter values, referred to as RA (return addresses), are integrated through tools like ltrace, ptrace, and ptrace. Additionally, library trace (ltrace) is employed to extract library function-related calls into the L set separately. The pre-processing stage focuses on identifying repeated addresses that occur in succession, which could lead to confusion. This redundant addressing is then removed, as repetitive system calls alone are not problematic; rather, consecutive calls can complicate the formation of the Return Address Path (RAP) and help mitigate issues related to Impossible Path Exploits (IPE).

3.1. Comparison Operation

In this, the method is adapted based on a probe-based approach where modules of attacker's snapshots are tested in non-legitimate mode, then after extracting updated datasets to define the kind of anomalies, illustrate a variety of attack possibilities, and organize these datasets into tables. Next, this will go for finding the variation by comparing the original datasets of training and the updated datasets of the testing phase generated from both modes of running.

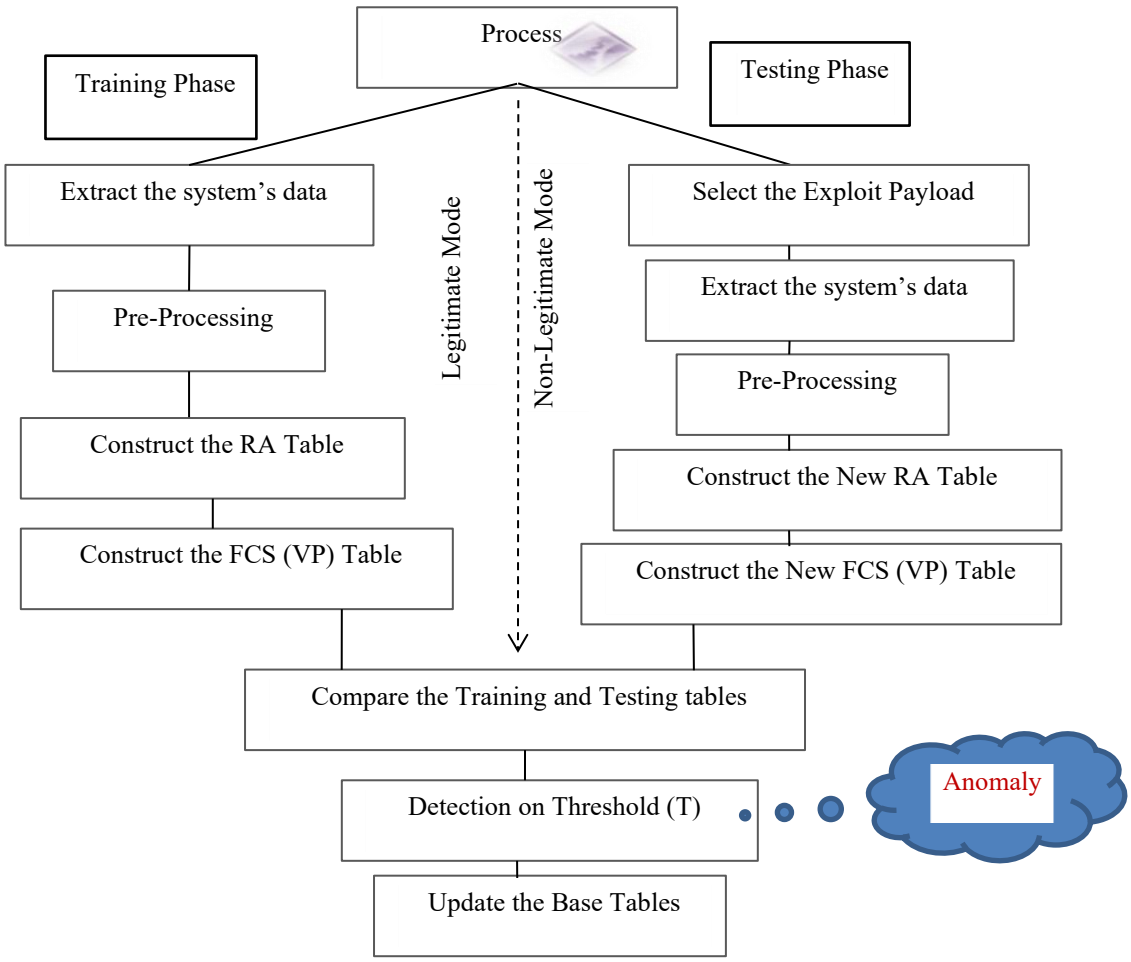

Fig. 2 Proposed ADS overview (Model-1)

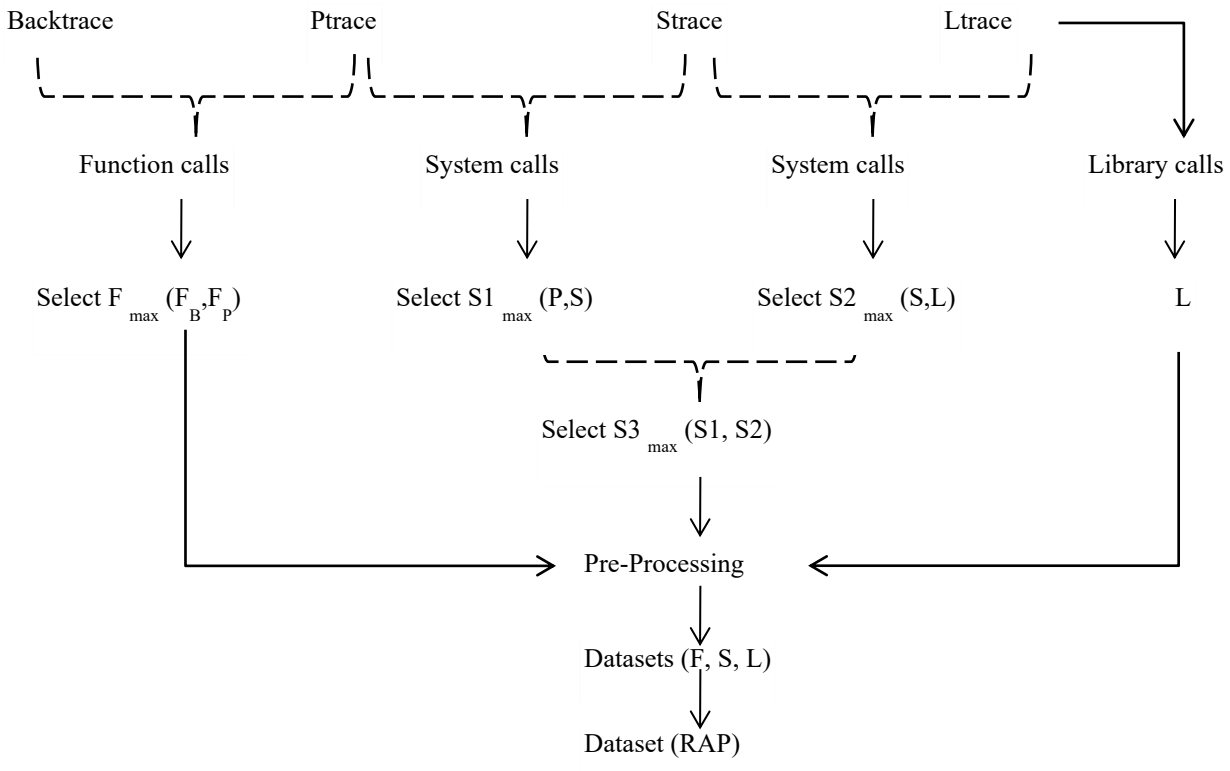

Fig. 3 Max ( ) function usage in datasets consolidation

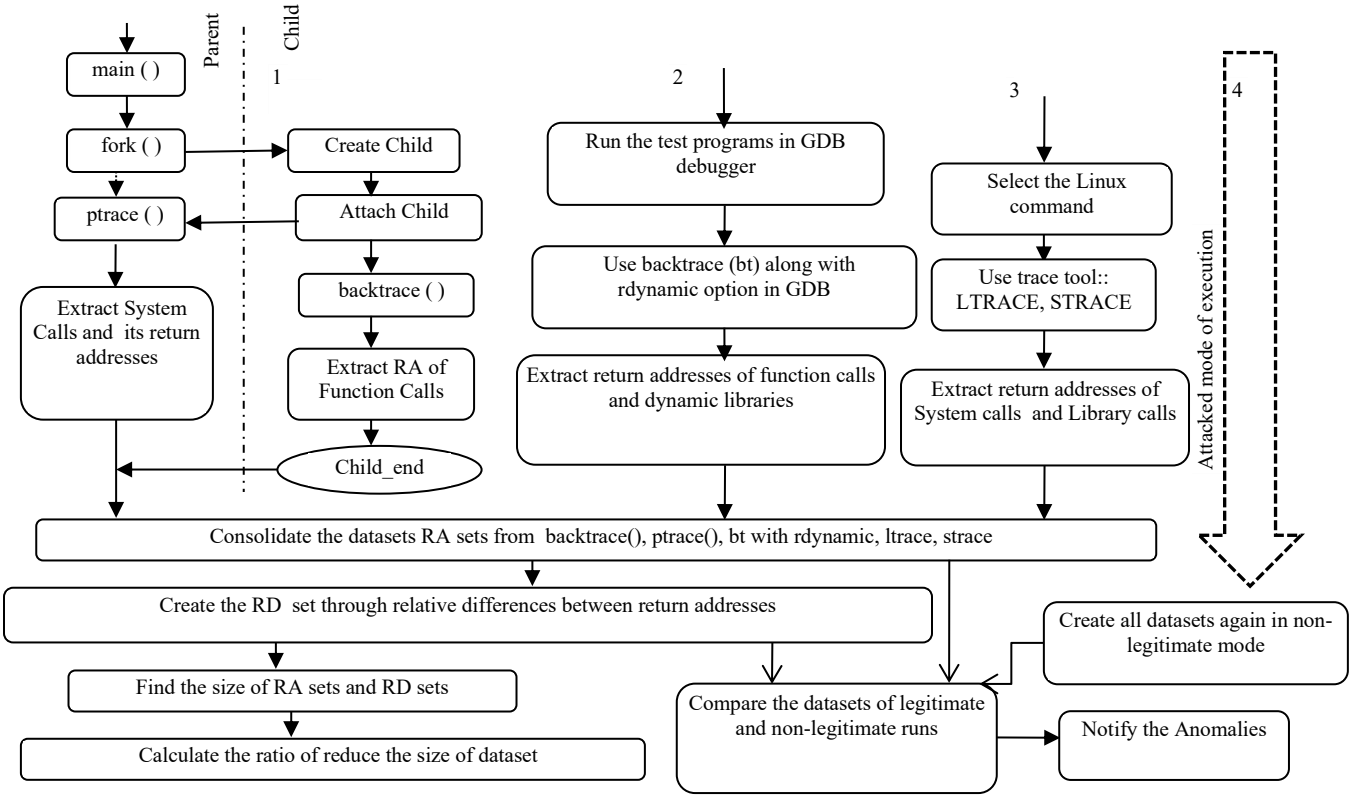

Fig. 4 The recommended prototype model-1 of the operational scheme, Datasets pulling out and RD set abnormalities are detected when a non-legitimate run (right 4 line) and (left 1, 2, 3 vertical lines supposed as authentic or legitimate)

For each iteration for each instance i=1 to k of the dataset from RAs extracted from phase during training, and likewise for each instance of j=1 to k updated from the dataset (RA set) during exploit mode in testing, it is indisputable to substitute R using the assortment variable F during the implementation.

$$M(R_i, R_j) = \sum_{k=1}^{n} D_k(R_i, R_j) \quad Where\ D_k(R_i, R_j) = \begin{cases} =0\ ,if\ R_i = R_j \\ \geq 1, if\ R_i \neq R_j \quad \therefore Anomaly \end{cases} \quad (1)$$

Where the term M signifies the functioning of the similarity (Match Function), R stands for a generated return address, the difference is annotated by D, which designates an alteration, and |D| depicts the threshold, where a value of zero is not considered an anomaly, while any non-zero value is treated as such. In comparing RAP datasets, it is decided to use the resemblance function from COSINE to assess the overall status of anomaly occurrences. This function also requires the bitwise XOR of the return addresses prior to the comparison. Next, both phases of the datasets are needed to match in order to show the evidence of anomalies by association. Here, with each i from 1 to n in the P1 (RAP set) recorded for the duration of training, we will then assess each i from 1 to n in the same line of the P2 (RAP set) through the detection line (testing).

$$COSINE(P1, P2) = \frac{P1 * P2}{||P1||.||P2||} = \frac{\sum_{i=1}^{n} P1_i * P2_i}{\sqrt{\sum_{i=1}^{n} P1^2} * \sqrt{\sum_{i=1}^{n} P2^2}} \quad (2)$$

Let us take an evaluation with an example of one of the outcomes:

RAP (P1) = {
==> 8049471 ==> 80493ec ==> 8049399 ==> 8049353 ==> 8048f14 ==> 8048f19 ==> 8049365 ==> 80493b3 ==> 80493f9 ==> 8049631 }
= {7, 4, 6, 0, 14, 3, 5, 12, 0, 1}

RAP (P2) = {
==> 8049471 ==> 80493ec ==> 8049399 ==> 8049353 ==> 8048f14 ==> 8048f19 ==> 8049365 ==> 80493b3 ==> 80493fe ==> 8049702 }
= {7, 4, 6, 0, 14, 3, 5, 12, 7, 0}

Then the evaluation of the cosine function is as follows

COS [P1,P2] =
(7*7+4*4+6*6+0*0+14*14+3*3+5*5+12*12+0*7+1*0)/
($(7^2+4^2+6^2+0^2+14^2+3^2+5^2+12^2+0^2+1^2)^{0.5}$ * $(7^2+4^2+6^2+0^2+14^2+3^2+5^2+12^2+7^2+0^2)^{0.5}$)

COS [P1,P2] = (475) / $((476)^{0.5} * (524)^{0.5})$

COS [P1,P2] = (475) / (21.817 * 22.891)

= 475/499.41 = 0.95

The interpretation of anomaly based on Cos (t) = 1 if it is a probable value, No RAP anomaly, else anomaly. The RAP's primary objective is to uncover exploits related to impossible paths and mimicry attacks. In the example above, an anomaly has been detected; however, the data shows a 95% similarity. This approach utilizes cosine similarity to verify the ratio of anomaly occurrences during testing.

3.2. Max ( ) Function and Pre-Processing

The Max() function plays a crucial role in validating lists of similar data gathered through various tracing techniques. Its primary purpose is to identify and select the list with the highest number of instances, which is referred to as Max. The instance count is demonstrated in Figure 3. In this proposed work, the Max() function is essential for maximizing the assortment and total of all calls mined from the process control running data; subsequently, it has been produced. Additionally, the Fmax(F_B, F_P) function has been utilized to enhance the number of return addresses for calling functions. Here, F_B represents the function calls obtained from the backtrace method, while F_P denotes the return addresses of the function calls derived from the ptrace() tracing technique.

In this article, the approach is more concise by taking various introspections and their combinations in command mode as well as experimental mode, shown in Figure 4, and also the hybrid mode of the above method, shown in Figure 5, and the adorned technique used with a combination of strace and LTTng for the extraction phase of the proposed model. This kind of contributions of experiments basically exhibits the internal behavior of cyber-physical systems to find the deviation from external attacks, system failures by vulnerable injection, and dynamic changes through external fraudulent command activities. The basic extraction tool STRACE utilizes the PTRACE concrete tool implicitly, but only the outcomes of traced data are different for each one. Similarly, LTTng operates externally on the Linux platform, serving as a lightweight tool with minimal complexity.

4. Proposed Experimental Model

The extended work with STARCE, excluding LTTng, is divided into two key phases, as illustrated in Figure 4. The training phase follows a sequence of steps aligned with the vertical lines labeled 1, 2, and 3. These steps include extracting system control data, consolidating the data, applying the Max () function for pre-processing, and constructing the datasets, which incorporate relative deviation (difference of addresses) calculations and hash tables for the RD set. The probing mode of detection starts with the fourth line to find the anomalies through injection or penetration kind exploits that were taken artificially.

This phase involves the integration of non-legitimate inputs, using the same vertical steps (1, 2, 3) from the training phase, albeit in the presence of an attack. During this phase, new datasets are created, hash tables are constructed, and both tables are compared to detect anomalies based on a set threshold. This methodology has been developed through a series of detailed steps outlined in the succeeding sections. The control data of the system is extracted using various

tracing options available on the Linux platform, as elaborated in Figure 3 and detailed in Figure 4. Specific tools, such as the rdynamic and 'bt' options available in the GNU Debugger (GDB) for tracing, are utilized in this task. While the tracing option 'bt' method is not unswervingly instigated in the mock-up programs, it also serves as a validation tool to ensure that the control data is extracted accurately from the simulated programs. The dynamic option is instrumental for tracing control data generated by dynamic calls during program execution. The data extraction includes register statistics, system call numbers, names, timing, and time complexity as part of system call information, as well as the return addresses of all calls and details of various other calls. For the purpose of this study, this work focuses on system call numbers and their corresponding return addresses. Data consolidation is achieved using the Max() functions, as discussed in the previous section.

In this process, similar data points are consolidated for all return addresses (RAs) associated with system calls and library calls. Function call addresses are inherently included in the RA set. However, separate datasets are created specifically for system calls and library calls. The peer group of system calls gathered from STARCE is verified with basic tracing options embedded in LINUX, and STRACE is a supplementary configured segment to extract the system calls with substantial performance, but only issues misfortune with additional overhead. The STRACE with LTTng hybrid model is presented as the intersection of a unique system calls collection shown in Figure 5, a basic ptrace collection is shown in Figure 6, and an additional exclusive trace collection is shown in Figure 7, separately from STRACE. As usual, the LTTng collection process has proceeded with babeltrace2 for analysis, including several steps depending on the user or kernel trace.

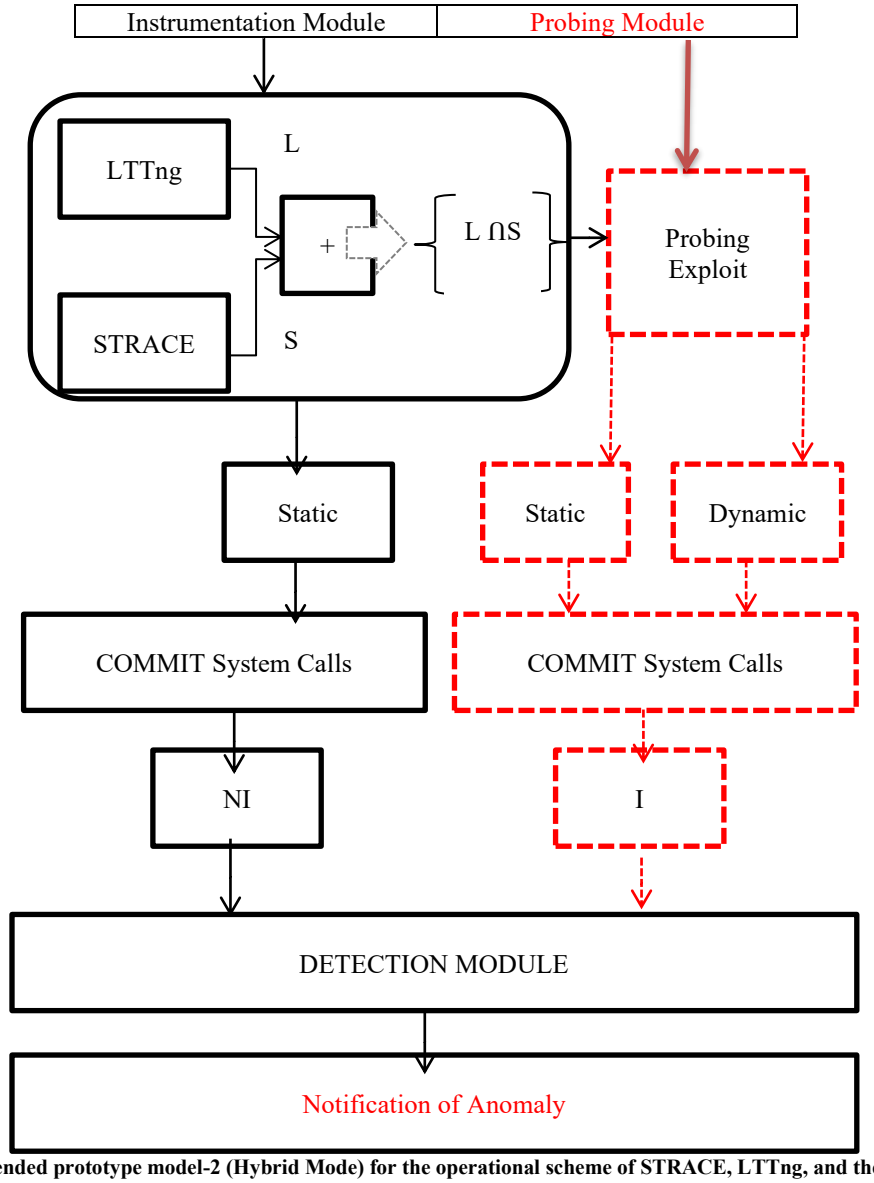

Fig. 5 Recommended prototype model-2 (Hybrid Mode) for the operational scheme of STRACE, LTTng, and their combination

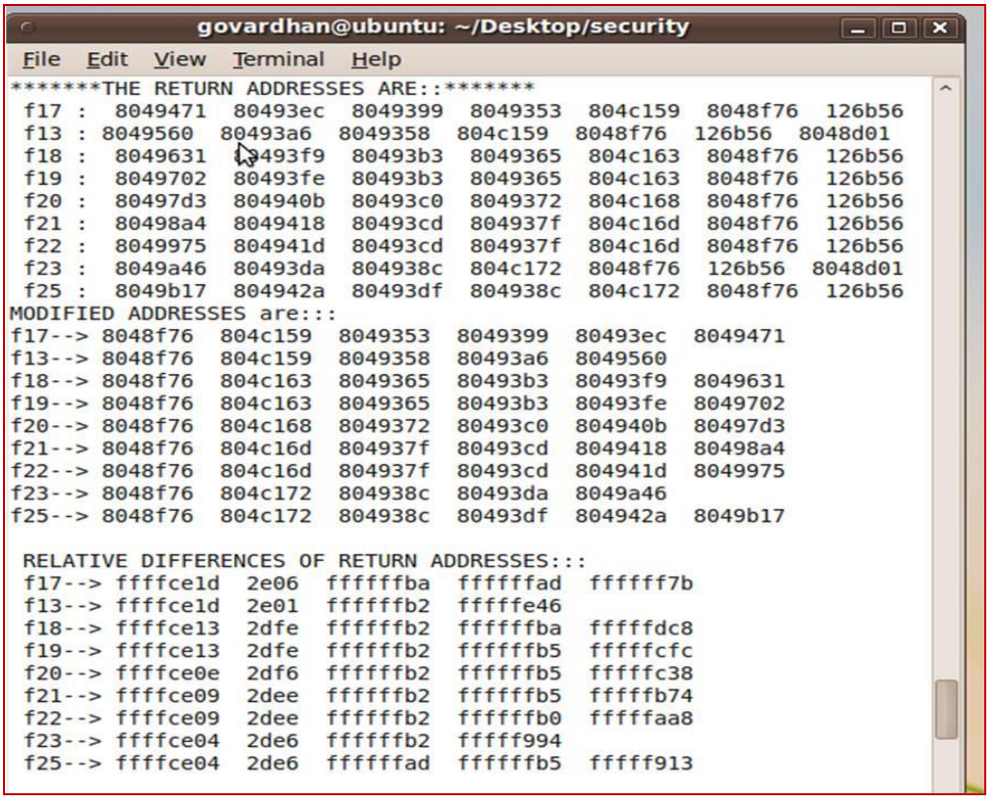

Fig. 6 The simulation program's output shows one of the snapshots on the Linux platform to demonstrate the RDs for Model-1

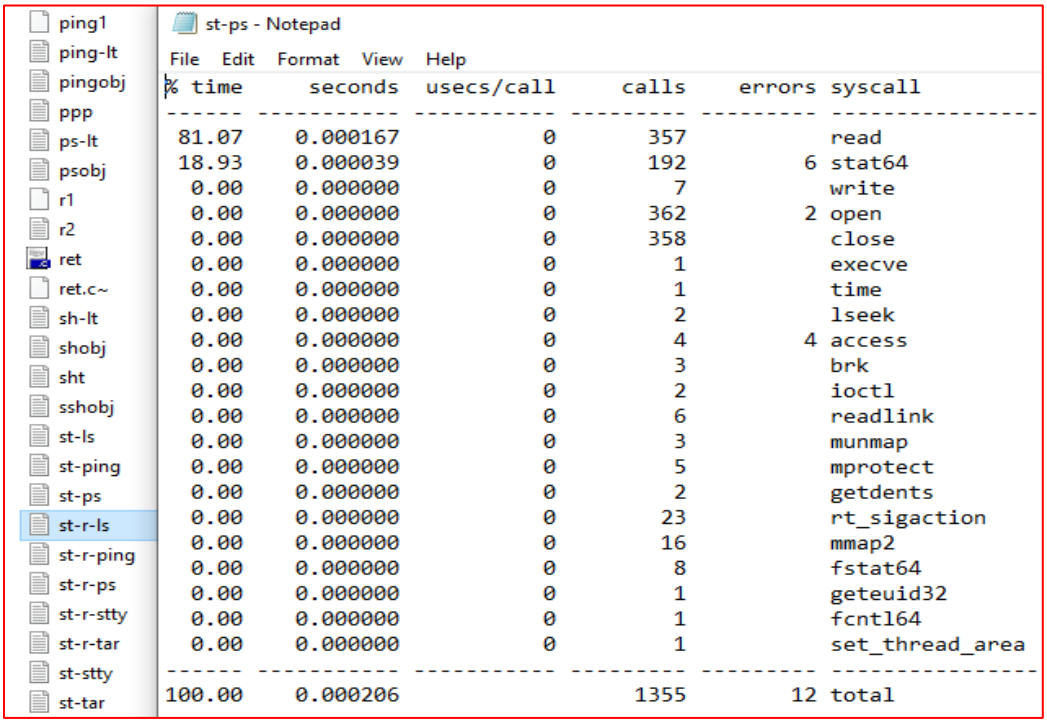

Fig. 7 Trace data statistics example for Model-1 (STRACE flow)

The pre-processing phase focuses on minimizing the control data within the system by pinpointing duplicate return addresses. These duplicates are then separated into distinct datasets for future application if necessary. This pre-processing approach offers a notable improvement over existing methodologies that simply rely on repeated system

call numbers within datasets for experimental purposes. The proposed method highlights that system calls can occur multiple times, basically through sibling-type utilities in Linux, entering kernel space from user-space with various return addresses. As a result, achieve a collection of unique return addresses in the datasets created. The only repetitions to be eliminated are from the main () and _init () functions in the extracted data. Once the key steps of pre-processing are completed, focus on the arrangement of the working structure for the finalized datasets. Further instrumentation procurement can generate uniqueness of RAs, which may be formed from the huge RA set initially, while keeping the SC set separate, and an exclusive LC set for next consolidations. The generative FCS or RAP paths are fabricated uniquely from a collection of tracing points (RAs) drawn from a stack of memory. Notably, the quantity of paths in sequence generated is kept at least count initially to test the evaluation, then follows the variation in sequence as thousands of return addresses can potentially create even more paths. To manage this complexity, the model focused on an average sequence size of 10 (i.e., grams of return addresses) initially extracted from the RA set, facilitating the construction of an optimally sized RD set with ease. The return addresses will form the Relative Differences (RD) resulting from the consolidated Return Address (RA) set. This finalized RD set plays a vital part in the testing and debugging stages of programs as well as applications. In this section, the model will elaborate in more detail about the method for preparing the RD set. The primary objective of the RD is to streamline the dataset, minimizing the excess timing computation that comes with generating thousands or even lakhs of control data points, which can be burdensome. Additionally, it is proposed to utilize hash construction for the RD set to further decrease the data size; however, this introduces a computational overhead. Consequently, the testing process will be conducted in parallel, comparing results. The procedure adapted in training for the recommended system tracks the steps in the mode of legitimate control in order from 1 to 3, allowing us to generate authentic datasets for the supplementary phase of testing. In contrast, the assessment of this perception transpires under a non-legitimate control mode using the mock-up programs, as well as taking Linux dynamic utilities sourced from commands. During this testing phase, new datasets will be created following the intervention of attacks on programs and applications, allowing deviations to be compared between the training and testing datasets, as previously explained. The initial phase of the proposed work involves the extraction of control data, illustrated in Figure 4, which provides an additional model to support thorough evaluation and implementation through methods such as rdynamic, system trace-STRACE (which details system calls), BACKTRACE (bt in GDB), for process tracing use PTRACE, and library trace LTRACE (to track system calls and library calls) all are fine recognized procedures for collecting system statistics on LINUX platforms. As shown in Figure 4, return addresses from all dropping calls are compiled from several tracing combinations and techniques, such as the backtrace method, which offers a subgroup of ptrace calls that contain both function calls and system calls. This scheme syndicates dynamic and BT (backtrace), which helps to mine the function calls of libraries that made entry into the stack drop an inline return addresses. The snapshot of collecting trace points is shown in Figure 6 and elaborates on the process of generating the RAs, modified RAs, and RDs for the evaluation of accurate results, and shows the finding procedure of RDs.

4.1. Relative Difference Between Unique RAs

The extraction of RAs from the call stack during the training phase needs to be mapped to their modified counterparts. These modified addresses accurately help compute the unique relative differences between the return addresses, which are consolidated uniquely.

Let us describe a group (set) of RAs as follows:
Return Address (RA) set =
{RA1, RA2, RA3, RA4, ..., RAn-1, RAn}.

From the above basics, find the RD set (Relative difference set) as follows:
RD set = {$|RA_1\text{-}RA_2|$,$|RA_2\text{-}RA_3|$,.....$|RA_i\text{-}R(A_j|$}
For corresponding pairs.

Here are 4 techniques to estimate the RDs.

1. From the 1st address generated (This Return address may be from _Init, or Main() functions):
   Ex: RD set= {$|RA_1\text{-}RA_2|$, $|RA_1\text{-}RA_3|$,.....$|RA_1\text{-}RA_j|$}.
2. By using fabricated (custom) RA: This is an input option provided by the user during input.
3. Ex: Input is assumed from User as UI_x= 0xf25f643f or 0xf25f643
   Compute RD set= {$|UI_x - RA_1|$,$|UI_x - RA_2|$,.....$|UI_x - RA_j|$}.
   By using a conceptual (base) address:
   Ex: Address Base is AB_1 = 0xffffffff or 0xffffff
   Compute new RD set=
   {$|AB_1 - RA_1|$,$|AB_1 - RA_2|$,...$|AB_1 - RA_j|$}.
4. Also by Consecutive addresses (Relative neighbor pair):
   Ex: Compute new RD set =
   {$|RA_1\text{-}RA_2|$, $|RA_2\text{-}RA_3|$,.....$|RA_i\text{-}RA_j|$}.

Finally, the RD set can be minimized as a required ratio by following the approach:

$$|\Delta| = \frac{S_D[S(RA_{set}) - S(RD_{set})]}{S(RA_{set})} * 100 \quad (3)$$

The value of $|\Delta|$ signifies the proportion adjustment in the RD set, by the relative difference set the RD set is signifying, S_D designates the variance factor among the RD set and the RA set, and the size is given by S. The process for computing the comparative variance and the proportion of deviations from the RA set to the RD set is outlined after the part one of

the Figure 4 shown in next Figure 6, with an illustrative example provided that shows the relative difference computed from the neighbor pair addresses transitioning from the RA set to the RD set, and vice versa. This section delves into the proposed work, detailing the algorithms, flowcharts, suggested models or techniques, and other associated efforts [1, 6]. The RD table contains the unique difference RDs, and further to reduce the size, a step was implemented to manipulate the part by incorporating a hash function (XOR) to the final RD table, and a new table H[RD] was constructed to store the hash digests basically in sequence, preferably in ascending order. This approach emphasizes the consideration of relative differences, offering an effective means to detect anomalies in malicious control data sourced from infected applications, particularly when integrating new addresses into memory segments.

The proposed hybrid work was recommended in Figure 5: here, the instrumentation model (Model-2) is separated from the probing module. These modules can work independently of each other, and comparison is extracting the invocation of anomalies. The instrumentation module focuses on the extraction of control information using STRACE and LTTng, where each can work in different lines, and a combination method is proposed in the intersection module to consolidate the exact number of system calls to be traced. It is said that uniquely, both cannot work together, but the introspection data can be consolidated to get more accurate outcomes. Generally, in static mode, it runs in non-intrusive mode to fix the list of system calls and RA data. At the same time, a probing module is performed in intrusive mode, either in static or dynamic, to conclude the exploited data, such as RA and SC control information. Thereafter, the model can notify the occurrences of anomalies using the module through detection. The algorithm is presented in the following lines for model-1, and similarly took exploits in combination of STRACE and LTTng, probably trace in a more intrusive manner, and finally commit the consolidation of system calls, then proceeds to test the process.

Algorithm: for Extracting Relative Differences from the output of basic Passtest:
1. Input: Set of Return Addresses (RA)
2. Convert: Transform the string representation of RA into integer format.
3. Select RD Type: Choose the specific type of relative difference to find from the options:
{FRA (First RA), CRA (Custom RA),
BRA (Base RA), RNRA (rel_NP_RA)}.
4. Invoke Function: relative difference ()` type to be invoked.
5. Perform Comparison: Execute the comparison process.
6. Return Value: Output the result in hexadecimal format using `Hex()`.
7. Repeat Process: Continue steps 2 through 6 for each RA.
8. Store Results: Save the calculated relative differences into the RD set.

Python algorithm for RD of return addresses from the output of passtest	
1	Ï←: RA set (Input);
2	Integer← Ï: string(RA);
	// Convert the String to Integer format
3	Γ← {FRA (First RA), CRA (Custom RA), BRA (Base RA), RNRA (rel_NP_RA)}.
	// RD type can be determined by selection
4	find_relative_difference ( );
	// exercise to find the difference between RAs
5	Φ← Phase-1- Compare ( );
6	H← Hex( ) // Form initial H: Values
7	Steps from 2 to 6 repeat.
8	Ṙ← Fix RD set
9	H(RD)← XOR (RD);
10	Φ2← Phase-2- Compare ( );

5. Results and Discussion

The work presented here is divided into three main sections. The first focuses on establishing the model and outlining the design methodology needed to accomplish the task. The second part discusses enhancements in Linux that facilitate faster operations and the creation of both training and testing datasets. Finally, the third section delves into anomaly classification based on test outcomes. The findings in this study stand out from prior research by emphasizing the importance of dataset size reduction while analyzing the comparison ratio of performance, such as True Positive Rate (TPR), Detection Rate (DR) of various models, and the actual implications of False Positive Rate (FPR) of identified probes, as well as real.

This comparison leverages the impact of mimicry attacks. In particular, the TPR (DR) measures how many interfering or unidentified Return Addresses (RAs) from the system calls, library calls, and also from function calls, (laterally with their ordering of sequence, and RDs) are accurately identified as abnormalities. Conversely, the False Positive Rate assesses the count of non-intrusive return addresses (RAPs) that are incorrectly flagged as abnormalities. Notably, the recommended scheme achieves an accurate DR of almost 100% with zero FPR, which denotes above-average performance; however, the false positive rate may increase when new addresses are introduced into memory segments.

This proposed approach incorporates a comparative analysis akin to earlier methods. The new RD dataset enhances anomaly detection, proving beneficial for various applications and program testing. The advantages of using return address and RD control data are illustrated through a case study on Linux utilities (PS) presented in the Table.1, particularly in the context of mimicry attacks. The experiments conducted here focus on standard and live system datasets collected from various sources to understand the triggering of anomaly detection, and also use the live system's data from virtual control. However, they approach the task as an offline method,

which poses a significant limitation for online applications currently being tested, such as servers and network systems. Notably, these methods face challenges from mimicry attacks, as summarized in Table 1. To highlight this, let us examine the first three methods that utilize the UNM or MIT dataset. In these cases, mimicry attacks are not adequately recognized, especially when contrasted with online methods that incorporate real datasets. For instance, consider a scenario involving an impression attack, also known as mimicry-based.

The demonstration of Mimicry Attacks is as follows:

The order or sequence of system calls is represented normally as

$S_R = \{R_1, R_2, R_3| R_4, R_5, R_6, R_7|R_8, R_9, R_{10}, R_{11}|\ldots\ldots\ldots\ldots\}$

Later, if any chances of attacks, it is possible to alter the structure and also do replicate the order, then it will turn identical to the original:

Interpretation of Attack: $R_i^n, R_j^n, R_k^{n-1}, R_l^n$

$\{R_4, R_5, R_6, R_7\} = \{(R_4, R_5, R_6, R_7), (R_4, R_5, R_6, R_7), (R_4, R_5, R_7), R_4, R_5, R_6, R_7\ldots R_4, R_5, R_6, R7\}$
(Or)
$\{ R_4,R_5, R_6, R_7\}= \{(R_4, R_5, R_6, R_7),(R_4, R_5, R_6, R_7, R_7), R_4, R_5, R_6, R_7\ldots\ldots R_4, R_5, R_6, R_7\}$

Where there R is a system call, the attackers are diverting the traditional sequence by appending the original old and legitimate sequence prefix or postfix with one false sequence. But sometimes this method is misclassified as FPR in case of new generation of system calls, maybe variation within version of OS, and dual kind system calls, such as IOCTL_RETVAL, IOCTL_ARGS, DUP, SETEGID, DUP2, SETGID, are mistakenly treated as anomalies and found misclassified (false positives) despite being non-intrusive. Meanwhile, certain intrusive system calls, such as OPEN_EBUST, OPEN_EEXIST, and specific file control calls like FCNTL_EAGAIN and FCNTL_EIO, may not be flagged as anomalies (false negatives).

Additionally, without a clear understanding of relative differences post-testing and debugging, there is a high risk of generating a 100% false positive rate. This relative difference remains constant regardless of the application's loading state within memory segments. The Model has been implemented in two ways separately; Model-1 is a mixing of powerful utilities verified at the command level, as well as programmed to consolidate the system introspection data. Many of the dynamic utilities were verified, but more elaborated on PS, along with a comparison of existing models developed based on benchmark and live datasets shown in Table.1. The traditional datasets took patterns of sequences formed with SC numbers, but live sequence patterns were not disclosed well.

Table 1. Comparative analysis on a case study on PS utility benchmark vs live with RD and without RD, and the effects of performance

S. no	Working Dataset to be tested	Contributed work and Year	System Calls Count W/o Siblings	System Calls Count With Siblings	False Alarms Count	FAR (FPR)%	Assaults (Attacks)	Description of Attacks and Positive Notes	* With proposed RD (Subsequent attack and Stuffing By Fresh EIP Address)
1	PS from UNM	[11] 1999	6144		0	0	Possible	Classified well, but seq. grams with mimicry attacks	TPR (ADR) =0 and FAR (FPR) =100%
2	PS from UNM	[6] 2001	10649		4505	42.3	Possible	More no. of attacks but Classified well	TPR (ADR) =0 and FAR (FPR) =100%
3	PS from LL (MIT)	[6] 2001	36088		996	2.7	Possible	Classified well	TPR (ADR) =0 and FAR (FPR) =100%
4	PS from MIT-LL	[9] 2006	4949		0	0	14 and Possible	Classified with FAR =0.0028, TPR= 100% for seq. Gram Mimicry attacks may have chances to exploit	This approach is close to the actual method, but only 12 sequences are shown, and DR=0 and FPR=100% with RD.

5	PS from UNM	[10] 2013	13399		Not Shown	10	Possible	Avg. of TPR and FAR shown, Misclassified. Possibility of Mimicry attacks.	DR=0 and FPR=100%
6	Virtual Live PS	Recommended Approach-1	1357	156	0	0.08	Shown	Only Probe allowed in User space, and signals and DLLs excluded	DR=99%, FPR<=0.9 %
7	Virtual Live PS	Recommended Approach-2	10318	5159	0	0	Shown in (I) Mode	In this, only Probe is allowed in User Space and estimates the possibility with fewer frequencies.	In NI Mode DR=100% and FPR=0% In I Mode DR=98% and FPR< = 2%

The existing developments are suffering from various scripting attacks, which may not update the calling sequence data, and found attacks also drop the data into an assumed legitimate mode due to the same RA, and also unnecessary system call data and system-level applications are burdened with the generation of huge addresses. Hence, the solution is adorned with a new concept of RD that may work accurately to overcome the scripting attacks and mimicry attacks.

The new approach-2 is a hybrid model giving the best results even in intrusive mode to disconnect the false alarms, even stuffing with new EIP, and also sensitivity (DR) is maximum with all thresholds. Also, it is found that maximum system calls and patterns are generated from siblings of existing, which may lead to false alarms, so uniqueness is best for minimizing the live dataset. The performance of the system is mainly dependent on the size of the datasets and convergence time during working on the training and testing modules, which are explained in Tables 2 and 3. Table 2 elaborates on the context of the size of generated data in original, authentic size in non-intrusive mode (NI), non-authentic (I-intrusive) size in intrusive mode, actual no. of system calls with siblings, and actual errors found. Both live datasets, RA and RD, are extracted through the I vs NI impact, which can be observed in the % change of data size from NI to I mode in order. Particularly, the PS utility has 10318 system calls in NI mode compared to 12227 in I mode by probing at the user space. Also, it is observed that GREP and FTP gives more change in file size due to the repetition of more siblings' calls. Table 3 presents the actual convergence times, including CPU time (in Seconds) and Memory Usage in MB, during Implementation (1: Strace, 2: LTTng).

The last column elaborates on the %C: Change using Strace and LTTng, as well as the combination of the intersection of models during the training and detection phases. The approximate changes are maximum for LTTng due to the application tool compared to the command-level STRACE. The recommended hybrid model is showing worth in finding less convergence time compared to similar developments done in the past. Table 4 shows the performance comparison of various utilities on the Linux live platform (a: sensitivity-detection rate, b: False Alarm rate in %) calculated overall up to finding the RD in non-malicious mode (NI-Non Intrusive), and similar detection phase outcomes are depicted in Table 5 for Intrusive mode (Non-legitimate) by elaborating DR, FAR (FPR), along with accuracy, and pictorial comparison is presented in Figure 8, which displays the accurate outcomes by visually hybrid model plot showing best with live dataset results.

Table 2. Legitimate and non-legitimate comparison of RD with file size change ratio using combination Models (%change: during actual training and testing in order)

Name of Utility	File size in Original	File Size in Authentic	Non-Legitimate	No. of System Calls (L) (NI)	No. of System Calls (NL) or (I)	No. of Errors	No of RAs	RD (L)	RD (NL)	% Change
Netstat	1112 KB	32 KB	37 KB	112	224	8	214	24 KB	28 KB	15 & 16
Ping	4908 KB	73 KB	89 KB	243	486	9	271	61 KB	69 KB	22&11
TTY	741KB	3KB	5KB	106	182	5	134	1.5KB	2KB	66&33
PS	30186 KB	2046 KB	2200 KB	10318	12227	327	13547	1746KB	1725KB	24& 26
Tar	665 KB	112 KB	128 KB	390	425	24	408	97 KB	111 KB	15&19
LS	813KB	14KB	22KB	156	204	4	214	2.7KB	4KB	57&48
Grep	876 KB	14KB	25KB	178	208	5	231	2.7KB	4KB	78&48
Ftp	16510KB	14KB	24KB	610	782	4	726	3KB	5KB	71&66

Table 3. Estimated convergence time, CPU time (in Seconds), and memory usage in MB during Implementation (1: Strace, 2: LTTng), last column elaborates the %C: change using strace and LTTng, and its combination of intersection of models during training and detection phase

Phase/ Method	Netstat		Ping		TTY		PS		TAR		LS		GREP		FTP		% C		%C (S^Lt)
Method	1	2	1	2	1	2	1	2	1	2	1	2	1	2	1	2	1	2	1and 2
CPU Usage	0.0016	0.002	0.020	0.029	0.0031	0.0035	0.030	0.0316	0.0019	0.0025	0.001	0	0.00002	0.00002	0.0051	0.0058	26	31	28% approx.
Memory Usage	2087	2147	2187	3076	1886	2156	5012	5316	4853	5287	4846	5261	1987	1857	3213	3720	41	48	44% change in approx.

Table 4. Performance comparison of various dynamic utilities on Linux live platform (a: sensitivity-detection rate, b: false alarm rate in %) calculated overall up to finding the RD in non-malicious mode (NI-Non Intrusive)

Method/Utility	Netstat		Ping		TTY		PS		TAR		LS		GREP		FTP	
Parameter→	a	b	a	b	a	b	a	b	a	b	a	b	a	b	a	b
STRACE	100	0	100	0	100	0	100	0	98	7	100	0	100	0	100	0
LTTng	100	0	100	0	100	0	100	0	99	4	100	0	100	0	100	0
Hybrid Mode	100	0	100	0	100	0	100	0	99	5	100	0	100	0	100	0

Table 5. Performance comparison of various utilities on Linux live platform (a: Sensitivity-detection rate, b: false alarm rate in %) calculated in overall up to finding the RD in malicious mode (I-Intrusive), generally user space introspection (* Combination)

Method/Utility	Netstat		Ping		TTY		PS		TAR		LS		GREP		FTP	
Parameter→	a	b	a	b	a	b	a	b	a	b	a	b	a	b	a	b
STRACE	98	7	99	3	99	0	98	2	99	9	99	2	98	2	97	6
LTTng	99	5	99	2	100	0	99	2	99	8	99	2	99	1	98	3
Hybrid Mode*	99	5	99	2	100	0	9	2	99	8	99	2	99	2	98	4
Overall Accuracy*	100%		100%		100%		100%		100%		100%		100%		100%	

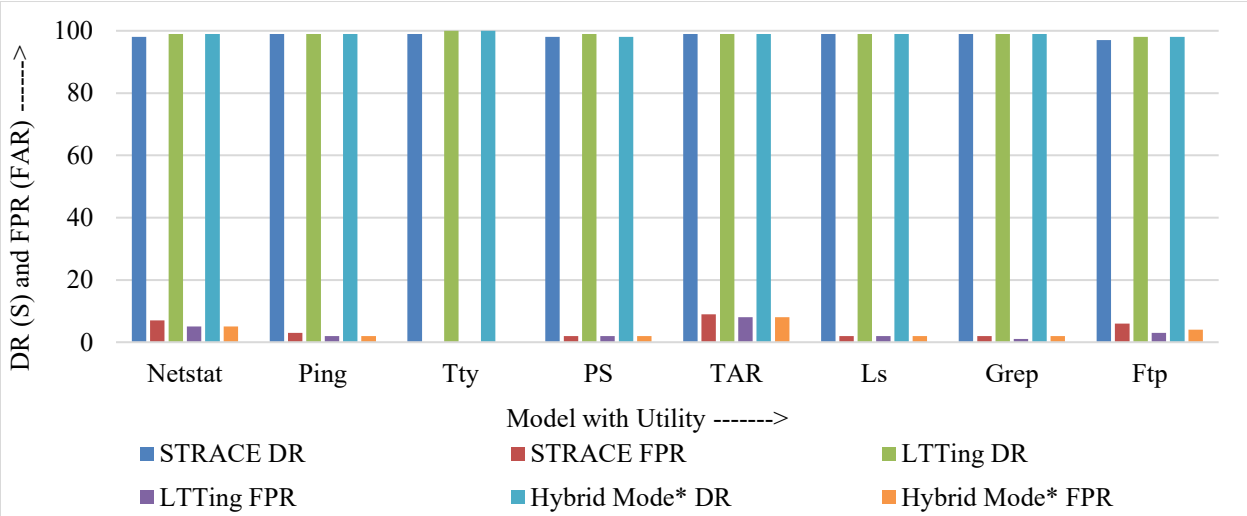

Fig. 8 The DR vs FPR for performance comparison of various dynamic utilities on the Linux live platform

5.1. Analysis of Results and Discussion

All the results with performance metrics are evaluated from trace findings, which were based on trace statistics shown in Figures 7 and 9. Further, all the results related to performance are to be explored well in the above tables with some statistical comparison of existing frameworks. The static mode of tracing system calls works offline and can help to find modern malware penetrations and subsequently analyze the potential dynamic anomalies in real-world applications, such as web-based based are closely facing the issue of ransomware attacks, which take control of the API and users' privileges, then expect ransom [15]. The main involvement of this theme is to investigate the performance indication of numerous permutations of tracing tools.

The proposed hybrid scheme exhibits unique RD, which is the best kind of criteria to find any kind of anomaly in the application execution, and found an injection of malicious code snippets, penetration threads, and abnormal payloads. From the hybrid model, we can also form the critical execution paths to find more anomalies, but it can form virtual paths by taking tracing return addresses from virtual address space, which may incur additional overhead to store the paths, and unique RDs are enough to enhance the performance shown in the above tables compared to the idea of critical path sequences [20]. The declarative events model encourages specifying synthetic events in a trace model of two different OS and exhibits the state information and performance of a real application, but lacks the point of anomalies [18]. This can be overcome by proposed models with unit anomaly by invoking a single RA or RD between two consecutive RAs. It is found that limited tracing platforms encouraged on heterogeneous systems without LTTng are represented in a lack of performance synchronization, and real-time embedded systems can also consume a process; also, trace synchronization and critical path extension are more difficult [19].

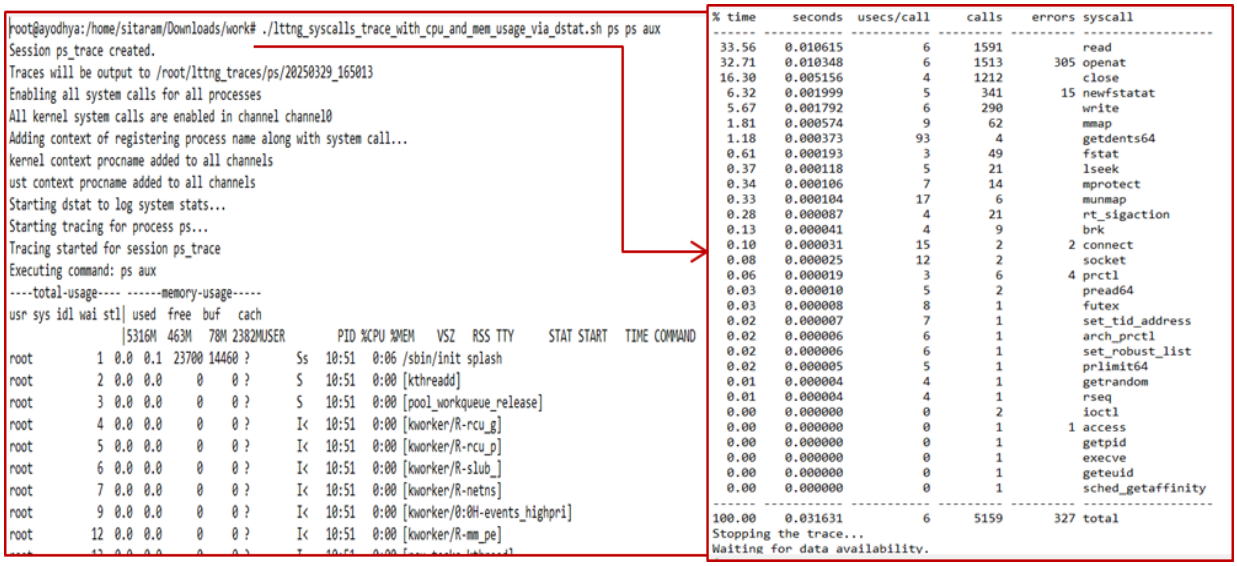

Fig. 9 Trace data statistics for Model-2 for exclusive LTTng

The statistical tracing and machine learning are implemented to detect the limited point anomalies, but the sequence of anomalies in further extension is not feasible in this method due to additional clustering overhead [21]. Hence, this paper is able to show exclusive performance on direct tracing with minimal consolidation of traced data, and enrichment of the LINUX platform can overcome the various kinds of anomalies observed in any kind of applications run on the present platforms. The developed combination is an enhanced scheme, as STRACE is quite fast due to its direct interaction with the kernel and is more suitable for validating notable performance and extracting the accurate semantics of system calls, which are quantified.

6. Conclusion and Further Enrichment

The proposed contribution introduces innovative ways to leverage the live control data from the system's run, including library calls, system calls, and function calls through an innovative perception known as the comparative variance among Return Addresses (RAs). This methodology proves to be invaluable for debugging and testing applications. Furthermore, it effectively reduces convergence time. By utilizing Linux containers, it can intelligently observe program execution behaviour and prototype the statistics collected from the background of systems as a significant feature of learning as dynamic knowledge. Lastly, a comparative analysis of two developments reveals that the strace tool has less overhead when paired with LTTng. Observations may find that the results of performance parameters may show that the variation depends on the algorithm, hardware, and method of utilizing the working tools.

From the past observations it is very difficult to trace the threats in real world applications entertainment, networks, system audits, social engineering, surveillances, transport tracking and management and many due to abnormal updates in structure of anomalies caused by bugs, updation failure, failure in networks, and payload errors, but in critical mode can able to trace errors or odd ones while in running of applications. So, for further improvement in performance, it is supposed to merge the machine learning adornments to LTTng and STRACE materials, which can enrich the outcomes for real-world applications. Also state that basic trace outcomes are modeled with LTTng state transition, and extension classification of anomalies through machine learning is supposed to be suggested to extract high-level performance for the dynamic context of real-time applications.

References

[1] H.H. Feng et al., "Anomaly Detection Using Call Stack Information," *2003 Symposium on Security and Privacy*, Berkeley, CA, USA, pp. 62-75, 2003. [CrossRef] [Google Scholar] [Publisher Link]

[2] R. Sekar et al., "A Fast Automaton-Based Method for Detecting Anomalous Program Behaviors," *Proceedings 2001 IEEE Symposium on Security and Privacy. S&P*, Oakland, CA, USA, pp. 144-155, 2001. [CrossRef] [Google Scholar] [Publisher Link]

[3] D. Wagner, and R. Dean, "Intrusion Detection via Static Analysis," *Proceedings 2001 IEEE Symposium on Security and Privacy. S&P 2001*, Oakland, CA, USA, pp. 156-168, 2001. [CrossRef] [Google Scholar] [Publisher Link]

[4] Debra Anderson, Thane Frivold, and Alfonso Valdes, "*Next Generation Intrusion Detection Expert System (NIDES): A Summary*," SRI International is an Independent, Nonprofit Corporation, pp. 1-47, 1995. [Google Scholar] [Publisher Link]

[5] Stephanie Forrest et al., "A Sense of Self for Unix Processes," *Proceedings 1996 IEEE Symposium on Security and Privacy*, Oakland, CA, USA, pp. 120-128, 1996. [CrossRef] [Google Scholar] [Publisher Link]

[6] Eleazar Eskin, Salvatore Stolfo, and Wenke Lee, "Modeling System Calls for Intrusion Detection with Dynamic Window Sizes," *Proceedings DARPA Information Survivability Conference and Exposition II. DISCEX'01*, Anaheim, CA, USA, pp. 165-175, 2001. [CrossRef] [Google Scholar] [Publisher Link]

[7] Surekha Mariam Varghese, and K. Poulose Jacob, "Anomaly Detection Using System Call Sequence Sets" *Journal of Software*, vol. 2, no. 6, pp 14-21, 2007. [Google Scholar] [Publisher Link]

[8] Sean Peisert et al., "Analysis of Computer Intrusions Using Sequences of Function Calls," *IEEE Transactions on Dependable and Secure Computing*, vol. 4, no. 2, pp. 137-150, 2007. [CrossRef] [Google Scholar] [Publisher Link]

[9] Darren Mutz et al., "Anomalous System Call Detection," *ACM Transactions on Information and System Security*, vol. 9, no. 1, pp. 61-93, 2006. [CrossRef] [Google Scholar] [Publisher Link]

[10] Syed Shariyar Murtaza et al., "A Host Based Anomaly Detection Approach by Representing System Calls as States of Kernel Modules," *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*, Pasadena, CA, USA, pp. 431-440, 2013. [CrossRef] [Google Scholar] [Publisher Link]

[11] C. Warrender, Stephanie Forrest, and Barak A. Pearlmutter, "Detecting Intrusions Using System Calls: Alternative Data Models" *Proceedings of the 1999 IEEE Symposium on Security and Privacy (Cat. No.99CB36344)*, Oakland, CA, USA, pp. 133-145, 1999. [CrossRef] [Google Scholar] [Publisher Link]

[12] Varun Chandola, Arindam Banerjee, and Vipin Kumar, "Anomaly Detection: A Survey," *ACM Computing Surveys (CSUR)*, vol. 41, no. 3, pp. 1-58, 2009. [CrossRef] [Google Scholar] [Publisher Link]

[13] Aleksandar Lazarevic, Vipin Kumar, and Jaideep Srivastava, *Intrusion Detection: A Survey*, Managing Cyber Threats, Springer, Boston, MA, pp 19-78, 2005. [CrossRef] [Google Scholar] [Publisher Link]

[14] Iman Kohyarnejadfard et al., "A Framework for Detecting System Performance Anomalies Using Tracing Data Analysis," *Entropy*, vol. 23, no. 8, pp. 1-24, 2021. [CrossRef] [Google Scholar] [Publisher Link]

[15] Thanh Nguyen, Meni Orenbach, and Ahmad Atamli, "Live System Call Trace Reconstruction on Linux," *Forensic Science International: Digital Investigation*, vol. 42, pp. 1-10, 2022. [CrossRef] [Google Scholar] [Publisher Link]

[16] Philippe Proulx, Tracing Bare-Metal Systems: A Multi-Core Story, The LTTng Project, 2014. [Online]. Available: https://lttng.org/blog/2014/11/25/tracing-bare-metal-systems/

[17] Mathieu Desnoyers, and Michel Dagenais, "Lttng: Tracing Across Execution Layers, from the Hypervisor to User-Space," *Linux Symposium*, Ottawa, Ontario Canada, vol. 1, pp. 101-106, 2008. [Google Scholar] [Publisher Link]

[18] Florian Wininger, Naser Ezzati-Jivan, and Michel R. Dagenais, "A Declarative Framework for Stateful Analysis of Execution Traces," *Software Quality Journal*, vol. 25, no. 1, pp. 201-229, 2016. [CrossRef] [Google Scholar] [Publisher Link]

[19] Thomas Bertauld, and Michel R. Dagenais, "Low-Level Trace Correlation on Heterogeneous Embedded Systems," *EURASIP Journal on Embedded Systems*, vol. 2017, no. 1, pp. 1-14, 2017. [CrossRef] [Google Scholar] [Publisher Link]

[20] Madeline Janecek, Naser Ezzati-Jivan, and Abdelwahab Hamou-Lhadj, "Performance Anomaly Detection through Sequence Alignment of System-Level Traces," *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*, New York, NY, United States, pp. 264-274, 2022. [CrossRef] [Google Scholar] [Publisher Link]

[21] Quentin Fournier et al., "Automatic Cause Detection of Performance Problems in Web Applications," *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, Berlin, Germany, pp. 398-405, 2019. [CrossRef] [Google Scholar] [Publisher Link]