# Explainable Machine Learning for API Call Sequence Analysis

SUBMISSION DATE / POSTED DATE

22-11-2022 / 07-12-2022

CITATION

Khan, Muhammad (2022): Explainable Machine Learning for API Call Sequence Analysis. TechRxiv. Preprint. https://doi.org/10.36227/techrxiv.21598998.v2

DOI

10.36227/techrxiv.21598998.v2

# Explainable Machine Learning for API Call Sequence Analysis

Muhammad Khan
Independent Researcher
mdk.md.ali.khan.786@gmail.com

**Abstract**—Although deep learning achieves state-of-the-art performance in tasks like vulnerability detection and classification, they have drawbacks. A significant disadvantage of deep learning methods is their inexplicability. Many deep learning models, especially sequential models like long short-term memories and gated recurrent units, operate as black-boxes. The outputs of these black-box models are uninterpretable by security analysts and software developers. This inexplicability of deep learning models hinders their acceptance in enterprises. It also prevents the knowledgeable system experts from removing the spurious correlations that the model might have learned. Thus, it is highly desirable to have explainable deep learning models to promote their acceptance in real-world systems. Another major drawback of deep learning is adversarial machine learning. Cyber attackers can utilize adversarial machine learning to fool the deep learning-based cyber-attack detectors. Prior research has shown that cyber-defenders can use explainable artificial intelligence to defend against adversarial machine learning attacks. Therefore, adding explainability to deep learning models for cyber-security is highly desirable. This article proposes a method that enhances the explainability of system-call sequence analysis-based vulnerability detection. Our method can potentially pinpoint the precise instruction calls that have triggered the vulnerability. This insight is valuable for the security analyst as he can now evaluate the sequence of system calls more efficiently.

## 1 Introduction

Organizations throughout the globe have been targets of adversarial cyber-campaigns in recent years. Hackers access millions of people's secret credentials and private information through malicious campaigns. In retrospect, cyber-researchers have discovered that most of these attacks were due to unpatched vulnerabilities and zero-day attacks [1]. Thus, it is necessary always to be vigilant and monitor our systems to detect vulnerability exploits.

Prior research has devoted significant efforts to detecting and classifying vulnerable systems. Sequential machine learning (ML) based models like long short-term memories (LSTMs), gated recurrent units (GRUs), and transformers have been the most successful in achieving state-of-the-art performance in these tasks. Other ML-based cyber-systems like intrusion and anomaly detection systems handle sequential data like network traffic and log files. Thus, sequential ML-based models are also used for these tasks.

All state-of-the-art sequence models like LSTMs, GRUs, and transformers have deep neural networks at the core of their architecture. However, a significant drawback of deep neural networks is their inexplicability. Researchers often refer to neural networks as black-boxes because it is challenging to reason their outputs. This lack of interpretability of neural networks limits their adoption in regulation-dominated applications like cyber-security and law [2].

Another major drawback of ML is the advent of adversarial ML. Adversarial ML is a relatively new technology that malicious actors can utilize to fool neural networks. This drawback of neural networks can potentially render neural network-based cyber-attack detectors ineffective. Prior research has exploited adversarial ML to minimally modify malware to evade state-of-the-art malware detectors while preserving the malicious nature of the malware. Wang et al. [3] have shown that developers can utilize explainable ML to analyze the weaknesses of ML-based malware. Interestingly, Marino et al. [4] have used adversarial ML to explain ML models of intrusion detection systems. These variegated insights are valuable for security analysts to generate vulnerability patches and be more vigilant.

Recent advancements in computer vision ML models' explainable artificial intelligence (XAI) have inspired researchers to investigate XAI in cybersecurity. For example, explainable graph neural networks (GNNs) for vulnerability detection have successfully shown that graph-specific explanations of vulnerability discovery GNNs provide deeper insights to security analysts than traditional methods [5]. Another work by Pirch et al. [6] demonstrates the use of XAI in vetting malware tags for better organization and categorization of malware. However, since XAI for vulnerability detection is a relatively developing area of research, efficient benchmarks are still not well-established for comparing various XAI models. Therefore, Warnecke et al. [7] propose novel criteria to evaluate XAI models in cybersecurity.

Traditional ML-based vulnerability detection models can predict if a sequence is malicious or benign. However, they cannot explain their decisions. Furthermore, the series of instruction calls obtained from execution traces are enormous, often amounting to millions of instruction calls. The large size of these sequences makes it difficult for the security analyst to analyze a potential threat predicted by the traditional ML model. Our framework mitigates the analysts' burden by pinpointing the exact elements of the sequence and their execution timestamps that trigger the exploit. Thus, the analyst does not have to analyze the entire exploit sequence. Instead, he can analyze only the sequence of exploit-triggering instruction calls predicted by our method. This significant reduction of search space mitigates the extent of human evaluation.

## 2 Background

This section discusses background material.

### 2.1 Seeker

Seeker is an anomaly detection system that uses artificial intelligence to efficiently detect abnormal system behavior. We design Seeker to classify instruction series at run-time efficiently. Unlike traditional methods, Seeker does not classify entire series of instruction calls. Instead, Seeker analyzes individual instruction calls and predicts the potential exploits they might trigger. Then, we use these predictions to update a dynamic state table at run-time. We raise the alarm when one or more states in the state table represent a signature. This combination of ML and non-ML-based techniques enables Seeker to be accurate and efficient at the same time.

First, we describe the high-level overview of the training pipeline. We begin by extracting the instruction series from the databases and the publicly available programs. Then, we convert the individual instruction calls into labeled feature vectors. Finally, we use these labeled feature vectors to train the Seeker classifier.

We describe the high-level overview of the Seeker inference pipeline next. First, we extract the instruction calls sequentially from the test program during inference. Then, we convert the current instruction call into a feature vector. The exploit classifier then analyzes this feature vector to predict the potential triggers that the current instruction call can execute. Finally, this list is input to the series checker. The series checker tracks the order of the instruction calls and raises the alarm if the current instruction executes a signature. If the current instruction does not implement a signature, we analyze the next instruction of the test program execution trace.

Since Seeker analyzes the individual instructions of the execution trace, it is easier to extract low-level insights. This low-level information about the particular series elements facilitates the explainability of Seeker's decisions.

#### *2.1.1 End-to-end inference pipeline*

We have designed Seeker to be capable of detecting exploits in real-time. Therefore, we input the executed instruction into our framework at run-time. We proceed to the next executed instruction if the executed instruction is present in our white-list of non-sensitive API calls. Otherwise, we analyze the instruction to determine if it triggers any undesirable characteristic. First, we obtain the feature vector for the executed instruction. The Seeker classifier processes this feature vector. Next, the classifier outputs the list of potential exploits that the executed instruction might trigger. If this list is empty, we terminate our analysis of the current instruction and proceed to the next executed instruction. Otherwise, we input the list to the Seeker series checker. The series checker has a dynamic state table that stores the states in our threat model. From the state table, we extract the states predicted by the classifier. Next, we compute the cosine similarities between the states of the extractions with the feature vector of the executed instruction. We can successfully detect semantically similar instructions by making this comparison. Finally, we update the exploit states similar to the feature vector of the executed instruction with the feature vectors of the following instruction in the corresponding exploit signatures. However, if the state table indicates that the executed instruction has implemented a potential signature, we stop the execution and raise the alarm. Otherwise, we proceed to analyze the next executed instruction.

#### *2.1.2 Series Checker*

Cyberattackers can exploit a vulnerability by executing a set of malicious instructions in a particular series. However, running the same set of instructions in a different order may not trigger the vulnerability [8, 9]. Thus, we need to track the series of program statements and instruction calls to detect exploits. Traditional series models, namely LSTMs and transformers, can track the order implicitly. However, this leads to high computation overheads. Therefore, we propose a state table-based approach to track series of program instructions.

First, we discuss a naive pattern matching approach that can be implemented with a state table [10, 11, 12]. Then, we discuss the drawbacks of this method and show how we can overcome these drawbacks with Seeker. Every row in the state table corresponds to a unique exploit. We can initialize the exploit states with the first instruction call in the respective exploit signatures for the naive approach. When we encounter any instruction call in the current state table during program execution, we update the corresponding states with the following instruction call of the exploit signature. If the encountered instruction call was the last one in the signature series(s), we would raise the alarm indicating that the API call has executed the corresponding exploit(s).

Although the naive method is much more lightweight than series models, it has two drawbacks that limit its applicability in real-world scenarios. First, the naive pattern matching approach cannot detect semantically-similar instruction calls [13]. As a result, the attacker may avoid detection by executing different APIs with the same functionality as the instruction in the signature. Second, checking the existence of every instruction in the state table is a time-intensive task. This timing overhead reduces the usefulness of naive pattern matching in real-time scenarios [8, 14].

Our solution, Seeker, addresses these drawbacks of naive pattern matching. Instead of storing the instructions in the state table, we store the feature vectors of the instruction calls. Then, we compare the feature vectors of the executed APIs with the entries in the state table. We use cosine similarity to measure the similarity between them. This method enables us to capture semantic similarities between APIs. If the cosine similarity is above a threshold value, we update the state table with the feature vector of the following API in the exploit chain. We experimentally observed that a threshold value of 0.9 enables the model to achieve the highest F1 score (note that the F1 score is a better performance measure than accuracy when the dataset is imbalanced).

To address the second drawback of considerable time overheads, we do not compare the feature vector of the executed API with all the states in the state table. Instead, we compare it with only the states of the categories predicted by the Seeker classifier. For example, if the Seeker classifier predicts that instruction call $i$ triggers 23 and 71, we compare the feature vector of $i$ with only the states 23 and 71. Our experiments show that this method requires nearly 10× fewer comparisons per instruction call.

## 3 Proposed Methodology

We describe our proposed methodology based on Seeker in this section. We aim to minimally modify the Seeker inference pipeline to integrate the explainability module in Seeker easily. We are also mindful that explainability does not deteriorate Seeker's efficiency.

Firstly, the class hierarchy of API calls makes it challenging to detect anomalous call sequences. We demonstrate an example of class hierarchy in Fig. 1. Our method of API call hierarchy extraction is based on the methods of malware detection by analyzing API call sequences [15, 16, 17].

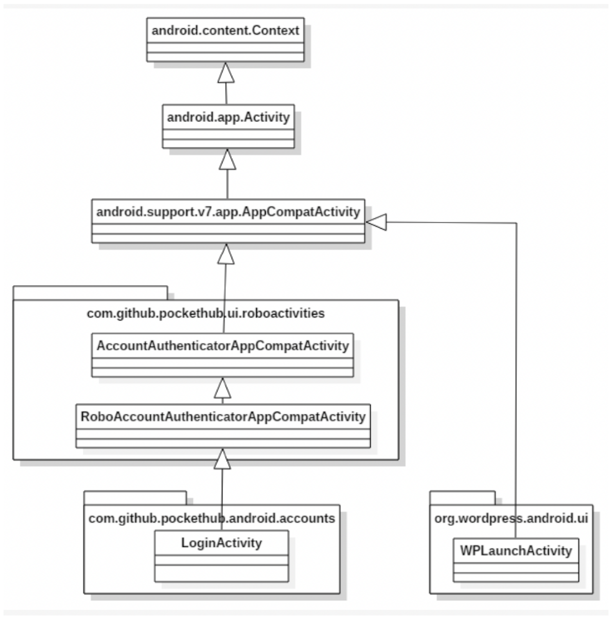
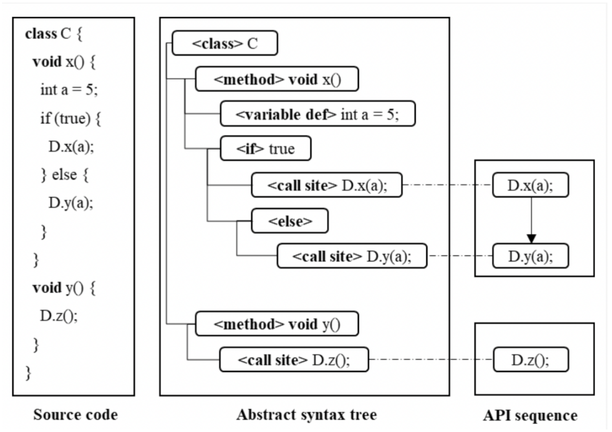

android.content.Context
android.app.Activity
android.support.v7.app.AppCompatActivity
com.github.pockethub.ui.roboactivities
AccountAuthenticatorAppCompatActivity
RoboAccountAuthenticatorAppCompatActivity
com.github.pockethub.android.accounts
LoginActivity
org.wordpress.android.ui
WPLaunchActivity

Fig. 1: An example of class hierarchy

However, the class hierarchy information can be extracted from source codes. Then, we generate the sequence of API calls. We demonstrate an examples of API call sequence extraction methodologies in Fig.2 and Fig.3.

We propose to create a state table for each signature. We update each of these state tables with the series indices of the potential instructions that may trigger the pothole. For example, let us consider that the instruction series $\{a, b, c\}$ can trigger $X$. Let us represent the instructions that are semantically similar to $a, b, c$ by $a_i, b_i, c_i$ respectively. We show an example of the proposed state table of $X$ in Table 1.

It is possible to construct a chain from the state table demonstrated in Table 1. We present the algorithm for creating the list of series from a state table in Algorithm 1.



```
class C {
  void x() {
    int a = 5;
    if (true) {
      D.x(a);
    } else {
      D.y(a);
    }
  }
  void y() {
    D.z();
  }
}
```

Fig. 2: An example of API sequence extraction from source code

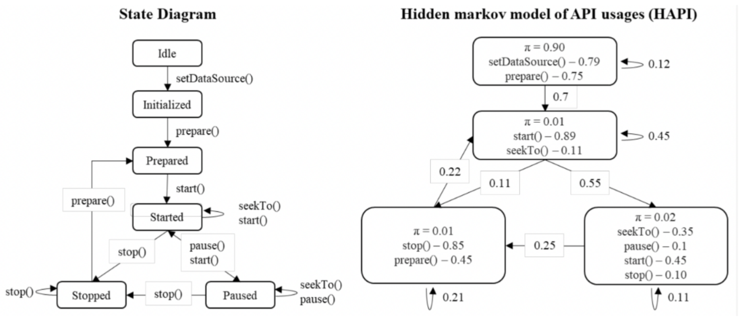


Fig. 3: An example of API call sequence extraction from call graphs

As shown in Algorithm 1, there are few constraints for generating a series from a state table. The constraints are as follows.

1) The $signID$ of an instruction determines its relative position in the signature. If an instruction with a lower $signID$ executes after an instruction with a higher $signID$, it is not triggered.
2) The generated trace should have instruction calls that have a strictly increasing order of instruction $seqID$'s. This property establishes the feasibility.

An example of the potential series extracted from Table 1 are $\{a_1, b_1, c_1\}$, $\{a_1, b_1, c_2\}$, $\{a_1, b_1, c_3\}$, and $\{a_1, b_2, c_3\}$.

The analysts can thus examine only the sub-series output by our method instead of the entire series. This minimized

TABLE 1: An example of a state table of $X$ with signature $\{a, b, c\}$. $p_i$ refers to an instruction semantically identical to $p$.

| signID | instrName | execInstr | seqID |
|---|---|---|---|
| 0 | start | start | 0 |
| 1 | $a$ | $a_1$ | 17 |
| | | $a_2$ | 108 |
| 2 | $b$ | $b_1$ | 32 |
| | | $b_2$ | 83 |
| 3 | $c$ | $c_1$ | 25 |
| | | $c_2$ | 77 |
| | | $c_3$ | 97 |

```
Function $exploitSeqGen$ ($currInstr$, $indexSeq$,
  $instrSeq$, $st$) :
    Data: Instruction of signature being analyzed from
          state table $currInstr$; Set of potential series in
          terms of execution trace indices $indexSeq$; Set
          of potential series in terms of instruction calls
          $instrSeq$; State table $st$
    Result: Set of potential series in terms of execution
            trace indices; Set of potential series in terms
            of instruction calls
    for i = 0; i ¡ len(st); i++ do
        if st[currInstr][seqID][i] ¿ indexSeq[-1 then
            instrSeq.append(st[currInstr][execInstr];
            indexSeq.append(st[currInstr][seqID][i]);
            if curr == len(st) then
                return indexSeq, instrSeq;
            else
                return exploitSeqGen(currInstr+1,
                  indexSeq, InstrSeq, st;
            end
        else
            return indexSeq.append(-1),
              instrSeq.append("-1");
        end
    end
$indicesSeqSet, instrSeqSet =$
  $exploitSeqGen(1, [0], [start], stateTable)$
for j=0; j ¡ len(indicesSeqSet); j++ do
    if indicesSeqSet[j][-1] == -1 then
        delete indicesSeqSet[j];
        delete instrSeqSet[j];
    end
end
```

**ALGORITHM 1:** Series generation for a category from its state table

search space reduces the burden on the analysts. It also provides insights into the model's working, thus developing the trust of the practitioners in the model. Finally, the analysts can detect errors and spurious correlations, thereby correcting them. For example, if Seeker incorrectly predicts an instruction $v$ to be semantically similar to $w$, the analyst can make a note of it. Thus, he can ignore such spurious predictions in the future. This ignorance of spurious correlations makes the model more robust with the mechanism of human-in-the-loop.

## 4 Conclusion

In this article, we have proposed a methodology to extend Seeker to incorporate the explainability of its decisions with minimal overhead. We believe that the explainability of Seeker's decisions will increase the trust of developers in deploying Seeker in enterprise systems. Explainability also makes it feasible to include the analysts in the loop to make Seeker more robust.

## References

[1] V. Sehwag and T. Saha, "TV-PUF: a fast lightweight analog physical unclonable function," in *2016 IEEE International Symposium on Nanoelectronic and Information Systems (iNIS)*. IEEE, 2016, pp. 182–186.

[2] A. Patil, A. Wadekar, T. Gupta, R. Vijan, and F. Kazi, "Explainable lstm model for anomaly detection in hdfs log file using layerwise relevance propagation," in *IEEE Bombay Section Signature Conference*. IEEE, 2019, pp. 1–6.

[3] W. Wang, R. Sun, T. Dong, S. Li, M. Xue, G. Tyson, and H. Zhu, "Exposing weaknesses of malware detectors with explainability-guided evasion attacks," 2021.

[4] D. L. Marino, C. S. Wickramasinghe, and M. Manic, "An adversarial approach for explainable ai in intrusion detection systems," in *IECON 44th Annual Conference of the IEEE Industrial Electronics Society*. IEEE, 2018, pp. 3237–3243.

[5] T. Ganz, M. Härterich, A. Warnecke, and K. Rieck, "Explaining graph neural networks for vulnerability discovery," in *14th ACM Workshop on Artificial Intelligence and Security*, 2021, pp. 145–156.

[6] L. Pirch, A. Warnecke, C. Wressnegger, and K. Rieck, "Tagvet: Vetting malware tags using explainable machine learning," in *14th European Workshop on Systems Security*, 2021, pp. 34–40.

[7] A. Warnecke, D. Arp, C. Wressnegger, and K. Rieck, "Evaluating explanation methods for deep learning in computer security," in *5th IEEE European Symposium on Security and Privacy*, 2020.

[8] T. Saha, N. Aaraj, N. Ajjarapu, and N. K. Jha, "Sharks: Smart hacking approaches for risk scanning in internet-of-things and cyber-physical systems based on machine learning," *IEEE Transactions on Emerging Topics in Computing*, 2021.

[9] T. Saha, N. Aaraj, and N. K. Jha, "Machine learning assisted security analysis of 5g-network-connected systems," *IEEE Transactions on Emerging Topics in Computing*, 2022.

[10] F. Kerschbaum and N. Oertel, "Privacy-preserving pattern matching for anomaly detection in rfid anti-counterfeiting," in *International Workshop on Radio Frequency Identification: Security and Privacy Issues*. Springer, 2010, pp. 124–137.

[11] S. Y. Lim and A. Jones, "Network anomaly detection system: The state of art of network behaviour analysis," in *2008 International Conference on Convergence and Hybrid Information Technology*. IEEE, 2008, pp. 459–465.

[12] Z. A. Baig, "On the use of pattern matching for rapid anomaly detection in smart grid infrastructures," in *2011 IEEE International Conference on Smart Grid Communications (SmartGridComm)*. IEEE, 2011, pp. 214–219.

[13] J. Brown, T. Saha, and N. K. Jha, "Gravitas: Graphical reticulated attack vectors for internet-of-things aggregate security," *IEEE Transactions on Emerging Topics in Computing*, 2021.

[14] T. Saha, N. Aaraj, and N. K. Jha, "System and method for security in internet-of-things and cyber-physical systems based on machine learning," Jun. 23 2022, US Patent App. 17/603,453.

[15] R. Sihwail, K. Omar, and K. Z. Ariffin, "A survey on malware analysis techniques: Static, dynamic, hybrid and memory analysis," *Int. J. Adv. Sci. Eng. Inf. Technol*, vol. 8, no. 4-2, pp. 1662–1671, 2018.

[16] D.-J. Wu, C.-H. Mao, T.-E. Wei, H.-M. Lee, and K.-P. Wu, "Droidmat: Android malware detection through manifest and api calls tracing," in *2012 Seventh Asia joint conference on information security*. IEEE, 2012, pp.

62–69.

[17] A. Pektaş and T. Acarman, “Deep learning for effective android malware detection using api call graph embeddings,” *Soft Computing*, vol. 24, no. 2, pp. 1027–1043, 2020.