SHARKS: Smart Hacking Approaches for RisK Scanning in Internet-of-Things and Cyber-Physical Systems based on Machine Learning

Tanujay Saha, Najwa Aaraj, Neel Ajjarapu and Niraj K. Jha *(Fellow, IEEE)*

Abstract—Cyber-physical systems (CPS) and Internet-of-Things (IoT) devices are increasingly being deployed across multiple functionalities, ranging from healthcare devices and wearables to critical infrastructures, e.g., nuclear power plants, autonomous vehicles, smart cities, and smart homes. These devices are inherently not secure across their comprehensive software, hardware, and network stacks, thus presenting a large attack surface that can be exploited by hackers. In this article, we present an innovative technique for detecting *unknown* system vulnerabilities, managing these vulnerabilities, and improving incident response when such vulnerabilities are exploited. The novelty of this approach lies in extracting intelligence from known real-world CPS/IoT attacks, representing them in the form of regular expressions, and employing machine learning (ML) techniques on this ensemble of regular expressions to generate new attack vectors and security vulnerabilities. Our results show that 10 new attack vectors and 122 new vulnerability exploits can be successfully generated that have the potential to exploit a CPS or an IoT ecosystem. The ML methodology achieves an accuracy of 97.4% and enables us to predict these attacks efficiently with an 87.2% reduction in the search space. We demonstrate the application of our method to the hacking of the in-vehicle network of a connected car. To defend against the known attacks and possible novel exploits, we discuss a defense-in-depth mechanism for various classes of attacks and the classification of data targeted by such attacks. This defense mechanism optimizes the cost of security measures based on the sensitivity of the protected resource, thus incentivizing its adoption in real-world CPS/IoT by cybersecurity practitioners.

Index Terms—Artificial Intelligence; Attack Graphs; Cyber-Physical Systems; Cybersecurity; Embedded Systems; Internet-of-Things; Machine Learning.

1 Introduction

Cyber-physical systems (CPS) use sensors to feed data to computing elements that monitor and control physical systems and use actuators to elicit desired changes in the environment. Internet-of-Things (IoT) enables diverse, uniquely identifiable, and resource-constrained devices (sensors, processing elements, actuators) to exchange data through the Internet and optimize desired processes. CPS/IoT have a plethora of applications, like smart cities [1], smart healthcare [2], smart homes, nuclear plants, smart grids [3], autonomous vehicles [4], and in various other domains. With recent advances in CPS/IoT-facilitating technologies like machine learning (ML), cloud computing, and 5G communication systems [5], CPS/IoT are likely to have an even more widespread impact in the near future.

An unfortunate consequence of integrating multiple devices in an ecosystem is the dramatic increase in its attack surface. Most of the CPS/IoT devices are energy-constrained, which makes them unable to implement existing elaborate cryptographic protocols and primitives as well as other conventional security measures across the software, hardware, and network stacks [6, 7]. The diverse range of embedded devices in the network and inherent vulnerabilities in the design and implementation, coupled with an absence of standard cryptographic primitives and network security protocols, make CPS/IoT a favorable playground for malicious attackers. Although lightweight cryptographic protocols [8, 9] and hardware-based (lightweight) authentication protocols [10, 11] mitigate some threats, most of the vulnerabilities remain unaddressed. Another challenge in securing CPS/IoT is the large amount of accessible data generated by the numerous communication channels among devices. Such data, in the absence of adequate cryptographic technologies, pose a threat to the CPS/IoT device and consequently impact user privacy, data confidentiality, and integrity. Moreover, CPS/IoT are vulnerable to a plethora of attacks [6, 12], e.g., buffer overflow exploits, race conditions, XSS attacks that target known vulnerabilities, and new (undiscovered) vulnerabilities, the exploit of which is referred to as a zero-day attack.

In this article, we propose an ML-based approach to systematically generate new exploits in a CPS/IoT framework. We call this approach SHARKS, which is an acronym for Smart Hacking Approaches for RisK Scanning. ML has already found use in CPS/IoT cybersecurity [13, 14, 15],

• *T. Saha, N. Ajjarapu and N. K. Jha are with the Department of Electrical Engineering, Princeton University, New Jersey, NJ, 08544 ({tsaha,najjarapu,jha}@princeton.edu. N. Aaraj is with Technology Innovation Institute, UAE (najwa@tii.ae).*

primarily in network intrusion and anomaly detection systems [16]. These systems execute ML algorithms on data generated by network logs and communication channels. In the methodology that we propose, ML instead operates at both system and user levels to predict unknown exploits against CPS/IoT.

SHARKS is based on developments along two important directions. Recognizing the need to depart from the traditional approaches to cybersecurity, we observe that the main objective of many security attacks on CPS/IoT is to modify the behavior of the end-system to cause unsafe operations. Based on this insight, we propose to model the behavior of CPS/IoT under attack, at the system and network levels, use ML to discover a more exhaustive potential attack space, and then map it to a defense space. Our approach enables a preemptive analysis of vulnerabilities across a large variety of devices by detecting new attacks and deploying patches ahead of time.

We analyze an exhaustive set of real-world CPS/IoT attacks that have been documented and represent them as regular expressions. An ML algorithm is then trained with these regular expressions. The trained ML model can predict the feasibility of a new attack. The vulnerability exploits predicted to be highly feasible by the ML algorithm are reported as novel exploits. This approach successfully generated 122 novel exploits and 10 unexploited attack vectors. To demonstrate the applicability of our approach, we evaluate the trained model on the in-vehicle network of a connected car. The model was successful in discovering 67 vulnerability exploits in the car network.

The novelty of the proposed methodology lies in:

- Representation of real-world CPS/IoT attacks in the form of regular expressions and control-data flow graphs (CDFGs), where both control flow and data invariants are instrumented at the system level.
- Creation of an aggregated attack directed acyclic graph (DAG) with an ensemble of such regular expressions.
- Use of an ML model trained with these regular expressions to generate novel exploits in a given CPS/IoT framework.

The article is organized as follows. Section 2 provides a summary of the work that has been done in the application of ML and automation to cybersecurity. Section 3 discusses background material. Section 4 gives details of our methodology and the results obtained with it. Section 5 describes the application of our algorithm to a connected vehicle. Section 6 proposes a tiered-security framework, composed of defense DAGs, for protection against security vulnerabilities. Section 7 concludes the article.

2 Related Work

In this section, we discuss some of the major works that have been done to automate security for real-world threat mitigation. Many major classes of security vulnerabilities, like memory corruption bugs and network intrusion vulnerabilities, can be detected using automation techniques. The domain of cybersecurity and embedded security that has been highly influenced by the popularity of ML is intrusion detection systems (IDSs), in particular an IDS targeted at network-level attacks. Prior to the rapid advancements in ML, IDSs consisted of signature-based methods and anomaly-based techniques to detect intrusions in the network or the host systems. Proposed IDSs perform quite well but have their drawbacks. Signature-based methods require regular updates of the software and are unable to detect zero-day vulnerabilities. Anomaly-based methods can detect zero-day vulnerabilities but have a very high false alarm rate (FAR). The advent of ML alleviated some of these drawbacks and thus ML was widely adopted in IDSs. Researchers have used a wide variety of ML methodologies to tackle this problem, such as unsupervised learning [17], artificial neural networks [18, 19], Bayesian networks [20, 21], clustering methods [22, 23, 24], decision trees [25, 26], ensemble learning like random forests [27, 28], hidden Markov models [29], and support vector machines (SVMs) [30, 31]. More advanced deep learning based IDSs use generative adversarial networks [32] and autoencoders [33]. These methods provide a reactive security mechanism for detecting ongoing attacks. They also require significant computational overhead because the models need to be continuously trained on recent data and all incoming traffic must be processed by the ML model before it can be catered to by the system. Our method differs from these methods in that it provides proactive security and requires zero run-time overhead.

ML has recently been used for malware and rootkit detection on mobile devices [34, 35]. These methods analyze the application programming interface (API) call logs to detect malicious behavior. Another ML-based malware detection method analyzes the hardware performance counters (HPCs) to detect malware execution at run-time [36]. A drawback of ML systems that are trained on API call logs, HPCs, and network logs, is that they are only able to detect the types of malware they have been trained on. A novel malware with completely different behavior and signature will go undetected by these detectors. They would also face difficulties in detecting the same malware running on a different platform and operating system. Our method, on the other hand, trains on application-independent representations of system-level attacks, which makes it platform-independent and equips it with the intelligence to detect a much broader class of security breaches.

Attack graphs have been widely used for analyzing the security of systems and networks [37]. Generating attack graphs has been a longstanding challenge due to the state explosion problem. Various automation techniques, like model checking [38], rule-based artificial intelligence, and ML [39], have been used to tackle this challenge. Analysis of the attack graphs is also a challenge due to the enormous size and complexity of the graphs. Graph-based neural networks [40] and reinforcement learning [41] have been used to analyze attack graphs to detect vulnerabilities. This article uses attack graphs at a higher granularity to detect vulnerabilities and the exploits thereof across the entire hardware, software, and network stacks of CPS/IoT. In previous works, system-specific attack graphs have been used for vulnerability analysis. In this article, we propose a generalized attack graph that can be applied to detect vulnerabilities (and exploits thereof) in any CPS/IoT. We

buttress this claim by applying our approach to detect vulnerabilities in the in-vehicle network of a connected car.

Memory corruption bugs have been a longstanding vulnerability in computer systems. A detailed analysis of this problem is provided in [42]. Automation attempts have also been made for detecting such bugs. In [43], static analysis is used to detect memory corruption vulnerabilities.

The discovery of hardware vulnerabilities like Spectre [44] and Meltdown [45] in 2018 opened the door to new classes of side-channel attacks on device microarchitecture. An automated side-channel vulnerability detection technique for microarchitectures is proposed in [46]. Our method aims to achieve a similar goal, but across the entire hardware, software, and network stacks.

3 Background

We model existing CPS/IoT attacks as regular expressions and CDFGs. We train a popular ML model, namely SVM, with these CDFGs to predict new vulnerability exploits. This section provides an introduction to regular expressions, CDFGs, and SVM models that is required for ease of comprehending the rest of the article.

3.1 Regular Expressions

A regular expression is used to denote a set of string patterns. We use regular expressions to represent known CPS/IoT attacks in a compact and coherent manner.

The set of all possible characters permissible in a regular expression is referred to as its alphabet Σ. The basic operations permitted in regular expressions are [47]:

- **Set union**: This represents the set union of two regular expressions. For example, if expression A denotes $\{xy, z\}$ and B denotes $\{xy, r, pq\}$, then expression $A + B$ denotes $\{xy, z, r, pq\}$.
- **Concatenation**: This operation represents the set of strings obtained by attaching any string in the first expression with any string in the second expression. For example, if $A = \{xy, z\}$ and $B = \{r, pq\}$, then, $AB = \{xyr, xypq, zr, zpq\}$.
- **Kleene star**: A^* denotes the set of strings obtained by concatenating the strings in A any number of times. A^* also includes the null string λ. For example, if $A = \{xy, z\}$ then, $A^* = \{\lambda, xy, z, xyz, zxy, xyxy, zz, xyxyxy, xyzxy, ...\}$.

In this article, we define regular expressions at a higher granularity for the sake of generality. The symbols of the Alphabet Σ of our regular expressions are generic system operations. Σ = {"Access port 1234 of the system," "Overwrite pointer address during memory overflow," ..., "Download unwhitelisted malware"}. All the symbols used in our regular expressions can be found in the nodes of the attack graph presented later. These symbols constitute the regular expression alphabet Σ.

3.2 Control-data Flow Graph

The CDFG of a program is a graphical representation of all possible control paths and data dependencies that the program might encounter during its execution. The basic blocks of the program constitute the nodes of the CDFG. A basic block is a block of sequential statements that satisfy the following properties:

- The control flow enters only at the beginning of the block.
- The control flow leaves only at the end of the block.
- A block contains a data invariant or a low-level system call.

Basic blocks are widely used in areas like compiler construction and finite automata. Generally, basic blocks denote low-level computer instructions. For the sake of this article, we use basic blocks to represent higher-level instructions. We construct the CDFGs at the level of human-executable instructions rather than assembly-level instructions, as shown in Fig. 1. We do this to ensure generalizability of our method to a wide spectrum of applications and systems.

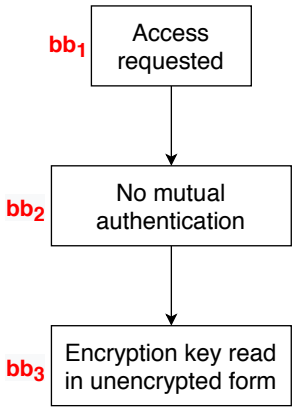


Fig. 1: Example of a CDFG

In Fig. 1, bb_1 denotes the action of requesting access to a device. Access can be requested through various ports and protocols like ftp, ssh or http. The port numbers and protocols to be targeted are system-dependent. We encapsulate these details into a single basic block to ensure generalizability of our method to all IoT systems and CPS. Similarly, in Fig. 1, bb_3 denotes the action of accessing the unencrypted key from the device key chain of a compromised, rooted or jailbroken device. The exact implementation is application-dependent because every application stores its keys at different locations. For example, WhatsApp (for both Android and iOS) stores its keys in the file /data/data/com.whatsapp/files/key. The assembly-level instructions for accessing the keys are not generalizable since they are application-dependent. Hence, we encapsulate all such instructions in a single basic block to facilitate generalization of our approach.

3.3 Support Vector Machine

We employ ML at the system level. Our training dataset does not have enough training examples to train a robust neural network. Thus, we use traditional ML approaches, instead of deep learning, for classification. Among traditional ML classification algorithms, SVM is one of the most robust classifiers that generalizes quite well.

SVM is a class of supervised ML algorithms that analyzes a labeled training dataset to perform either classification or regression [48]. It is capable of predicting the label of a new example with high accuracy. It is inherently designed to be a linear binary classifier. However, kernel transformations can be used to perform nonlinear classification as well. For a dataset with an n-dimensional feature space, a trained SVM model learns an $(n-1)$-dimensional hyperplane that serves as the *decision boundary*, also referred to as the *separating hyperplane*.

Many contemporary ML algorithms, e.g., k-nearest-neighbor classification, use a greedy search approach. However, SVM uses a quadratic optimization algorithm to output an optimal decision boundary. The two main limitations of SVM are its natural binding to binary classification and the need to specify (rather than learn) a kernel function.

4 Methodology

In our methodology, we extract intelligence from an ensemble of known CPS/IoT attacks and use this system-level adversarial intelligence to predict other possible exploits in a given CPS/IoT framework. The automated derivation of novel exploits and defenses broadly comprises extracting intelligence, discovering unexploited attack vectors, applying ML, and taking measures to secure the system. These processes are depicted in the flowchart of Fig. 2.

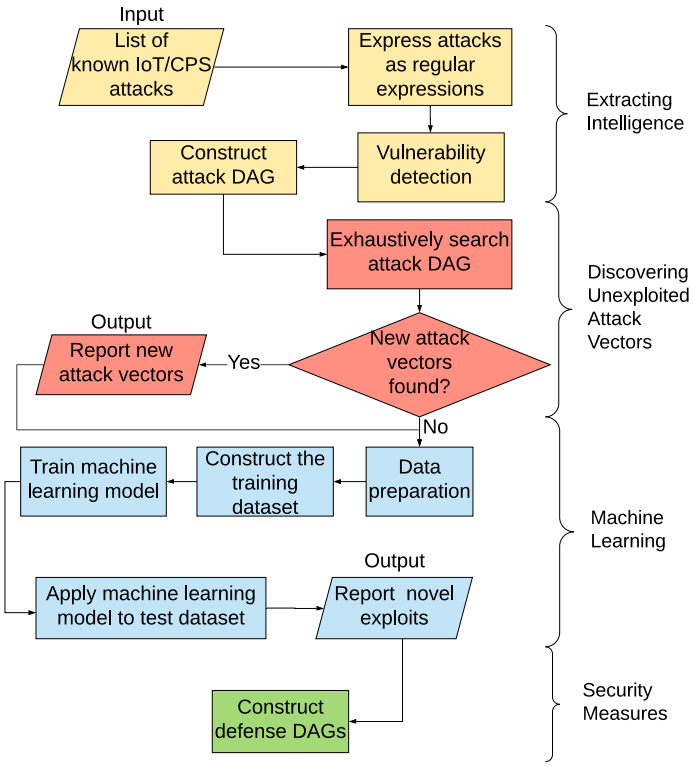


Fig. 2: Flowchart of the overall methodology

4.1 Extracting Intelligence

We document existing CPS/IoT attacks and decompose them into their constituent system-level actions and used data invariants. We use regular expressions to represent these constituent system-level operations. Then we combine the regular expressions of all the attacks to form an ensemble of interconnected system-level operations. This ensemble is represented as a DAG. This DAG is henceforth referred to as the aggregated attack DAG.

4.1.1 Data Collection

Next, we discuss how to extract knowledge from known attack patterns. To achieve this objective, we create a list of known CPS/IoT attacks. Then we classify these attacks into various categories based on the type of vulnerability being exploited. This list consists of 41 different attacks [6, 49, 50]. The most popular attacks among these and their regular expressions are shown in Table 1.

4.1.2 Data Transformation

In this phase, we decompose each attack into its basic system-level operations. We express these sequences of operations as regular expressions that are then represented as CDFGs. Each attack is now transformed into a CDFG with system-level operations as its basic blocks. The methodology of decomposing an attack into a CDFG is similar to the method used in [51].

An example of the data transformation procedure for a buffer overflow attack is given next. A buffer overflow attack can be expressed as a sequence of following actions:

1) dynamic memory allocation,
2) overflow of memory, and
3) frame pointer with overwritten memory.

Let bb_i denote the i^{th} basic block of the sequence. Then the corresponding regular expression is given by:

$$bb_i(\textit{Dynamic memory allocation})^*.bb_j(\textit{Overflow of memory}).$$
$$bb_k(\textit{Frame pointer with overwritten memory})$$

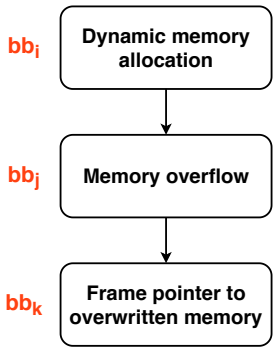


Fig. 3: CDFG of buffer overflow attacks

Here, bb_i denotes the dynamic memory allocation that occurs in the memory stack before a buffer overflow occurs. The Kleene star operation suggests that bb_i might be executed multiple times before bb_j is executed. Basic blocks bb_j and bb_k are similarly defined. Every basic block of the regular expression forms a node of the CDFG. The concatenation operation (represented by the dot operator) is represented by a branch from the preceding node of the concatenation operator to its succeeding node. The CDFG of a buffer overflow attack is shown in Fig. 3. In Fig. 3,

we can see that the concatenation between bb_i and bb_j is represented by a directed branch from bb_i to bb_j. The Kleene star operator should ideally be represented by a self-loop on the basic block. However, we ignore self-loops in our CDFG representation. This is because we combine these CDFGs into a DAG. The acyclic property of a DAG facilitates our further analysis, but the presence of self-loops violates this property.

4.1.3 Attack DAG

Every attack in our list is represented by its corresponding CDFG. All the CDFGs are combined to form a single DAG. This DAG, shown in Fig. 4, is our aggregated attack DAG. The attack DAG is a concise representation of the system and network-level operations of known categories of CPS/IoT attacks. Every path from a head node to a leaf node in the attack DAG corresponds to a unique attack vector.

We observe that certain basic blocks appear in multiple attacks. These basic blocks are represented as a single node in the attack DAG with in-degree and/or out-degree greater than 1. Our attack DAG has 37 nodes, represents 41 different attacks, and has a maximum depth of 6.

4.2 Applying Machine Learning

Once we have represented the known attacks in the attack DAG, we observe that some of its unconnected nodes can be linked together. Every new feasible link that is predicted by the ML model is considered to be a novel exploit of vulnerabilities. A link or branch is considered to be feasible if the control/data flow represented by that branch can be implemented in a real-world system. We use ML models to predict if directed branches between various pairs of nodes of the attack DAG are feasible. Manual verification of the feasibility of all possible branches in the attack DAG is too time-consuming. Let n be the number of nodes in the attack DAG and c be the number of examples in the training dataset. Then the size of the search space of possible branches is

$$\begin{aligned} 2\binom{n}{2} - c &= n(n-1) - c \\ &= n^2 - n - c \\ &= \Theta(n^2) \end{aligned} \quad (1)$$

This quadratic dependence makes it very expensive to perform manual checks to exhaustively examine the feasibility of all the possible branches. In our experiments, we show that using ML can reduce the search space by 87.2%.

We train the ML model using the attack DAG of known attack vectors. Once trained, it can predict the feasibility of new branches in the attack DAG. We derive an SVM model for this purpose. Since the dataset is very small, consisting of just 140 datapoints, it prevented us from being able to adequately train a neural network [52]. However, when our methodology is applied to a larger scope of cyberattacks, a neural network model might be an effective tool [53].

4.2.1 Data Preparation

We assign various attributes (features) to the basic blocks of the attack DAG depending on the type of impact the attack would have on the system and network. The exhaustive set of attributes that we used is composed of memory, data/database, security vulnerability, port/gateway, sensor, malware, authentication vulnerability, head node, leaf node, and mean depth of the node. Each node has a binary value (0 or 1) associated with every feature except the mean depth. The mean depth of a node denotes the average depth of the node in the attack DAG. For example, nodes *"Certificate proxying"* and *"SQL query with format -F"* have the attributes shown in Table 2.

We represent a branch in the attack DAG by an ordered pair of nodes, i.e., (*origin node*, *destination node*). The features of the branches of the attack DAG are required to train the ML model. The concatenation of the attributes of the origin and destination nodes represents the feature vector of a branch. For example, from Table 2, it can be observed that the feature vector for the node *"Certificate proxying"* is $[0, 0, 1, 0, 0, 0, 1, 0, 1, 1]$ and that of node *"SQL query with format -F"* is $[0, 1, 0, 0, 0, 0, 0, 0, 1, 3.75]$. A plausible branch from node *"Certificate proxying"* to node *"SQL query with format -F"* will be represented by the ordered concatenation of the feature vectors of the individual nodes, which is $[0, 0, 1, 0, 0, 0, 1, 0, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 1, 3.75]$. This vector represents the attributes of a plausible branch and serves as a datapoint for our ML model. The label of this particular datapoint is -1 because a certificate proxying vulnerability cannot be exploited to launch an SQL injection attack. This process of datapoint construction is illustrated in Fig. 5.

4.2.2 Training Dataset

Our SVM model learns from the underlying patterns that exist in known CPS/IoT attacks, some of which are shown in Table 1. This knowledge is encoded in the attack DAG. Thus, the training dataset is composed of all the existing branches (positive examples) and some infeasible branches (negative examples) of the attack DAG. The labels of the training dataset are:

- 1, if the branch exists in the attack DAG.
- -1, if a branch from the origin to the destination node is not feasible.

A negatively labeled branch denotes an impossible control/data flow. The detailed procedure for generating negative examples for the training set is discussed in Section 4.2.3.

Our training dataset consists of 140 examples, 39 of which have positive labels and the remaining have negative labels.

4.2.3 Negative Training Examples

Creating the set of negative training examples is more complicated than creating the set of positive training examples. This is because the absence of a branch in the DAG does not imply that the branch is infeasible. It implies that the branch was not exploited in any real-world IoT/CPS hack so far. We describe the process of finding negative examples next.

First, we classify the nodes of the DAG into broader vulnerability categories. These categories are:

- Memory vulnerability

TABLE 1: Real-world CPS/IoT attacks and regular expressions

Attack	Vulnerability category	Regular expression
Therac-25 Radiation Poisoning	Race condition / TOCTOU vulnerability	bb_i(access system call)*. bb_j(open system call)*
Ariane 5 Rocket Explosion	Integer overflow	bb_i(data invariant > max integer)*
Worcester Airport Control Tower Communication Hack	Buffer overflow	bb_i(dynamic memory allocation)*.bb_j(overflow of memory).bb_k(frame pointer with overwritten memory)
Bellingham, Washington, Pipeline Rupture	Buffer overflow	bb_i(dynamic memory allocation)*.bb_j(overflow of memory).bb_k(frame pointer with overwritten memory)
Maroochy Shire Wastewater Plant Compromised	Access control/Privilege escalation	bb_i(critical component with one-factor or one-man authentication)*
Davis-Besse Nuclear Power Plant Worm	Malware/Privilege escalation	bb_i(critical component with one-factor or one-man authentication)*
Worm Cripples CSX Transport System	Malware/Privilege escalation	bb_i(critical component with one-factor or one-man authentication)*
Worm Cripples Industrial Plants	Malware/Privilege escalation	bb_i(critical component with one-factor or one-man authentication)*
Browns Ferry Nuclear Plant	Distributed Denial of Service (DDoS) attack	bb_i(port traffic per second > threshold)
LA Traffic System Attack	DDoS attack	bb_i(data invariant > threshold)
Aurora Generator Test	Protocol vulnerability	bb_i(access requested)*.bb_j(no mutual authentication)*
Internet Attack on Epileptics	SQL injection	bb_i(user input)*.bb_j(user input not compliant with database format)
Turkish Oil Pipeline Rupture	Privilege escalation/ DDoS	bb_i(critical component with one-factor or one-man authentication)* + bb_j(data invariant > threshold)
Stuxnet Attack on Iranian Nuclear Power Facility	Malware through USB	bb_i(executive file of new executable at kernel level)*.bb_j(sending data through port to external C2)
Tests of Insulin Pumps	No authentication + No encryption Replay attacks	bb_i(transaction requested)*.bb_j(no time stamp check)*.bb_k(no mutual authentication)*.bb_l(no hash check)* bb_i(data in transit not encrypted)*
Houston, Texas, Water Distribution System Hack	Weak access management	bb_i(access requested)*.bb_j(no strong authentication, e.g., no public key infrastructure based authentication or two-factor authentication)*
Researcher Defeats Key Card Locks	No authentication	bb_i(access requested)*.bb_j(no mutual authentication)*.bb_k(encryption key read from memory in unencrypted format)*
Test of Traffic Vulnerabilities	Weak cryptographic measures	bb_i(no encryption of data/commands)*+(bb_j(no digital signature on sensor firmware)*. bb_k(illegal access through unobstructed port)*. (bb_l(reconfigure the system specs)* + (bb_m(access memory buffer). bb_n(overwrite allocated memory))*))
German Steel Mill Attack	Malware/Privilege escalation	bb_h(open downloaded file from spear-phishing email)*.bb_i(executive downloaded file from email)*.bb_j(critical component with one-factor or one-man authentication)*.bb_k(access business network)*.bb_l(access ports of entry to production network)*.bb_m(manipulate commands to the system)*
Fatal Military Aircraft Crash Linked to Software fault	Software fault	bb_i(access system files)*.bb_j(rewrite code for updates)*.bb_k(delete/modify important system files)*
Test of Smart Rifles	Weak password	bb_i(weak WiFi password)*.(bb_j(alter state variables)*+bb_k(gain root access))
Black Energy Ukrainian Power Grid Attack	Weak authentication	bb_i(spear phishing emails to access business network)*.bb_j(maneuver into the production network)*.(bb_k(erased critical files on disk) + bb_m(took control over important network nodes)*)
Mirai Botnet Attack	Weak authentication + DDoS	bb_i(weak password)*.bb_j(port traffic per second > threshold)
Unidentified Water Distribution Facility Hack	Web vulnerabilities	(bb_i(phishing emails to access credentials)*+bb_j(SQL injection attacks to get credentials)*).bb_k(weak storage of credentials on front-end server)
"WannaCry" Ransomware Attacks	Buffer overflow Cryptographic key management	bb_i(dynamic memory allocation)*.bb_j(overflow of memory)*.bb_k(frame pointer with overwritten memory in SMBv1 buffer)* bb_i(process starts encrypting data)*.bb_j(process new to the system and not whitelisted)*

TABLE 2: Node attributes

Attribute	Certificate proxying	SQL query with format -F
Memory	0	0
Data/Database	0	1
Security vulnerability	1	0
Port/Gateway	0	0
Sensor	0	0
Malware	0	0
Authentication vulnerability	1	0
Head node	0	0
Leaf node	1	1
Mean depth	1	3.75

- Network protocol vulnerability
- Weak cryptographic and authentication measures
- Malware
- Social engineering

We observe that some of these vulnerability classes are independent of each other. The pairs of independent categories are listed next:

1) Memory vulnerability; network protocol vulnerability
2) Memory vulnerability; social engineering
3) Network protocol vulnerability; social engineering
4) Weak cryptographic and authentication measures; malware with social engineering

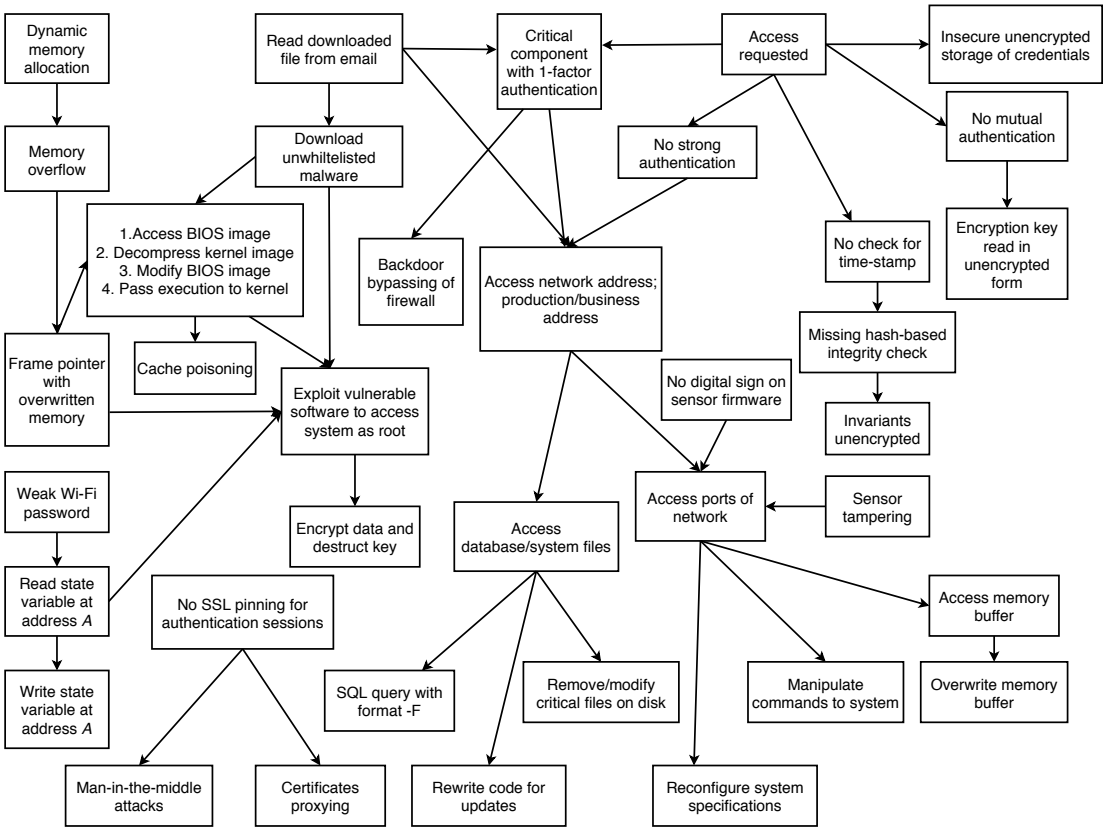


Fig. 4: The aggregated attack DAG

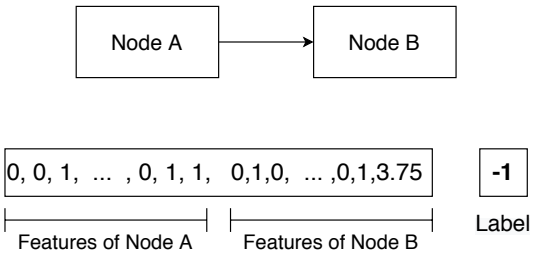


Fig. 5: Representing a plausible branch as a datapoint of the training or test dataset

Most of the nodes of the mutually independent categories will not have feasible branches between them. Such infeasible branches are added to the set of negative training examples. However, there may exist some exceptions in which branches between nodes of independent categories are feasible. For example, the nodes *"Overwrite allocated memory"* and *"Access ports of network"* belong to the mutually independent attack categories, namely *Memory vulnerability* and *Network protocol vulnerability*, respectively. However, there exist attack vectors in which the buffer overflow vulnerability can be exploited to obtain the device port assignments for the TCP protocol or to listen to network ports on the device. We carefully exclude such branches from our set of negative examples. Similarly, there exist a few infeasible branches between nodes of non-independent categories as well as between nodes of the same category. To include such negative branches into our training set, we experimentally observe certain statistical properties of the existing negative examples. Then we use these properties to filter out more negative examples. The observations are:

- Most branches from head nodes to the leaf nodes are infeasible.
- Most branches among leaf nodes are infeasible.
- The difference between the mean depths of the source and destination nodes of an infeasible branch is either smaller than -0.09 or greater than 2.0.
- Most infeasible branches have a high Hamming distance (HD) between the feature vectors of the source and destination nodes. The mean HD of feasible and infeasible branches were observed to be 2.93 and 4.30, respectively.

Filtering out the probable negative examples with these observations reduces our search space. Then we manually select the negative examples from this reduced search space.

4.2.4 Training

The ML model has multiple parameters that can be tuned to achieve optimal performance [54]. The parameters of the SVM model that we experimentally tuned during training are mentioned below.

1) **Regularization parameter (C):** Regularization is used in ML models to prevent overfitting of the model to the training data. Overfitting causes the model to perform well on the training dataset but poorly on the test dataset. This parameter needs to be fine-tuned to obtain optimal performance of the model. The value of C is inversely proportional to the strength of regularization.

2) **Kernel:** The kernel function transforms the input vector x_i to a higher-dimensional vector space $\phi(x_i)$, such that separability of inputs with different labels increases. We use the radial basis function (RBF) as our kernel function. The RBF kernel is defined as:

$$k(x_i, x_j) = exp(-\gamma||(x_i - x_j)||^2) \quad (2)$$

3) γ: Parameter γ defines how strong the influence of each training example is on the separating hyperplane. Higher (lower) values of γ denote a smaller (larger) circle of influence.
4) **Shrinking heuristic:** The shrinking heuristic is used to train the model faster. The performance of our model does not change in the absence of this heuristic.
5) **Tolerance:** The tolerance value determines the error margin that is tolerable during training. A higher tolerance value causes early stopping of the optimization process, resulting in a higher training error. A higher tolerance value also helps prevent overfitting.

The parameters of the SVM model are chosen such that we get zero false negatives (FNs). The parameters that have the greatest influence on accuracy are the regularization parameter (C), kernel of SVM, and γ. We performed a parameter search over various combinations of these parameters and plotted the number of FNs against them. The results of our parameter search experiments are shown in Fig. 6. In Fig. 6a, we observe that the RBF kernel yields the lowest number of FNs. We also observe that the number of FNs increases with an increase in the value of γ. It is very important that we obtain zero FN in order to regard all the negative predictions of the model as infeasible exploits. In Fig. 6b, we observe that only one combination of parameter values in our parameter search space gives us zero FNs. We choose these parameters for our SVM model, as shown in Table 3.

TABLE 3: SVM parameters

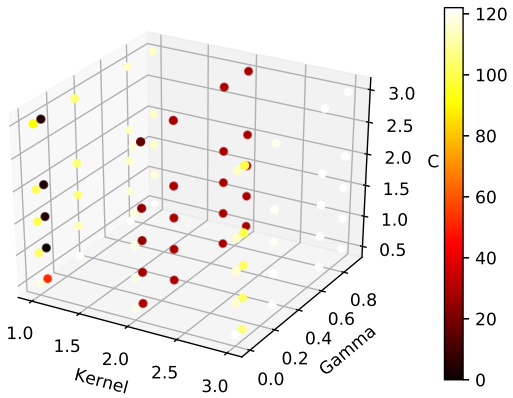
Parameter	Value
C	1.0
Kernel	RBF
γ	0.0556
Shrinking heuristic	Used
Tolerance for stopping	10^{-3}

4.2.5 Verification

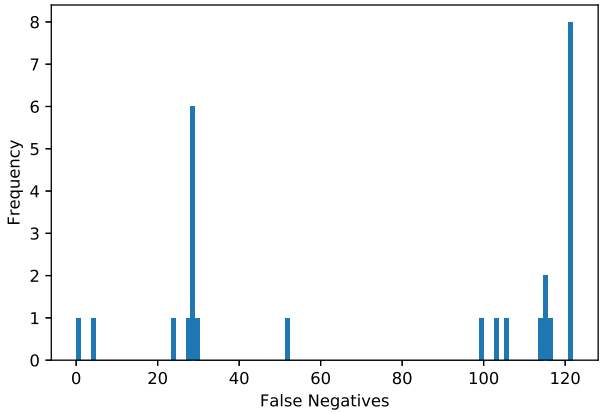
A test example is positive if the sequence of two basic blocks is a permissible control/data flow in a given system. Determining the control/data flow in a program is generally a hard task. However, in this article, we define the basic blocks at a human-interpretive level. This makes it easier for a human expert to determine if the sequence of basic blocks in the test example is feasible or not.

The SVM model predicts 153 positive labels out of 1192 test datapoints. A positive label indicates that the test datapoint is a potential novel exploit. Manual verification of all the 1192 datapoints in the test dataset revealed that 1161 predictions by the SVM model are accurate, resulting in a test accuracy of 97.4%.

The parameters of SHARKS were chosen to achieve zero FN. However, our SVM model outputs a few false positives (FPs). To eliminate these FPs, manual verification is necessary. In the absence of SHARKS, a human expert would have to verify all 1192 potential vulnerability exploits manually. With the assistance of SHARKS, it is sufficient to verify only the 153 positive predictions of the SVM model. Thus, SHARKS helps reduce the search space of possible novel exploits from 1192 to 153, which is an 87.2% reduction in manual checks.



(a)



(b)

Fig. 6: Parameter search: (a) 3D scatter plot showing the number of FNs for various combinations of parameters (Kernels 1, 2, and 3 refer to the RBF, polynomial, and sigmoid kernels, respectively), and (b) histogram of FN frequencies

4.3 Experimental Results

In this section, we present the experimental results. We begin by demonstrating why we chose an SVM model for novel exploit detection. In addition to SVM, we evaluated the following models: k-nearest neighbors (k-NN), naive Bayes, decision tree, and stochastic gradient descent (SGD) based linear SVM. We compare their accuracies, precision/recall values, false positive rates (FPR), and F1 scores in Table 4. It is clear that SVM (with C=1) performs the best.

TABLE 4: Performance of ML models

Model	Accuracy	Precision	Recall	FPR	F1
Decision Tree	86.8%	0.40	0.89	0.14	0.55
k-NN (k=2)	92.8%	0.60	0.62	0.04	0.61
k-NN (k=3)	92.0%	0.54	0.88	0.08	0.67
k-NN (k=4)	94.5%	0.70	0.70	0.03	0.70
k-NN (k=5)	93.0%	0.58	0.86	0.06	0.69
Naive Bayes	90.5%	0.46	0.26	0.03	0.34
SVM (C=1)	**97.4%**	**0.80**	**1.0**	**0.03**	**0.89**
Linear SVM with SGD	90.6%	0.49	0.75	0.08	0.59
SVM (C=2)	93.8%	0.60	0.97	0.06	0.76
SVM (C=3)	92.8%	0.56	0.96	0.08	0.71

We use the SVM model to predict the feasibility of all plausible branches of the attack DAG. The test dataset contains all plausible branches except the branches present in the training dataset. The branches are converted into test vectors by the method depicted in Fig. 5. The attack DAG has 37 nodes and the training set has 140 examples. Putting $n = 37$ and $c = 140$ in Eq. (1), we observe that the test dataset contains 1192 test vectors. The SVM model successfully predicts the existence of 122 new feasible branches in the attack DAG. Each new branch corresponds to a unique novel exploit.

Some of the 122 feasible branches of the attack DAG that were predicted by ML are listed in Table 5. These attacks have been chosen to represent the most popular vulnerability categories.

TABLE 5: Novel exploits discovered

Branch discovered	Vulnerability category
Read downloaded file from email ⟶ Overflow of memory	Buffer overflow
Access network ports ⟶ Encrypt data and destroy key	Privilege escalation
Access system files and databases ⟶ Reconfigure system specifications	Access control
Download unwhitelisted malware ⟶ Bypass firewall using backdoor	Malware
Access network address ⟶ Encryption key read from memory in unencrypted form	Cryptographic flaw
Critical component with 1-factor authentication ⟶ Access Basic Input/Output System (BIOS) image	BIOS boot level attack
Exploit malware to access system as root ⟶ Cache poisoning	Cache poisoning

The confusion matrix in Table 6 shows the number of true negatives (TNs), FPs, FNs, and true positives (TPs). The SVM model achieves zero FNs, which indicates that a negative prediction is always correct.

TABLE 6: Confusion matrix

N=1192	**Actual = No**	**Actual = Yes**	
Predicted = No	TNs = 1039	FNs = 0	1039
Predicted = Yes	FPs = 31	TPs = 122	153
	1070	122	

In Fig. 7, we categorize the novel exploits into six categories. We can see that access control vulnerabilities (including privilege escalation), weak cryptographic primitives, and network security flaws are most common vulnerabilities with high likelihood of exploit. We also observe that vulnerabilities with lower exploit likelihood are BIOS vulnerabilities and cache poisoning attacks. This is expected because a successful BIOS attack or a cache poisoning attack involves one or more of the following: boot-stage execution, shared resources with adversary, side-channel access, kernel code execution, and close proximity to the CPS/IoT devices at very specific time instances. These operations involve higher complexity in building the exploit chains across various system elements.

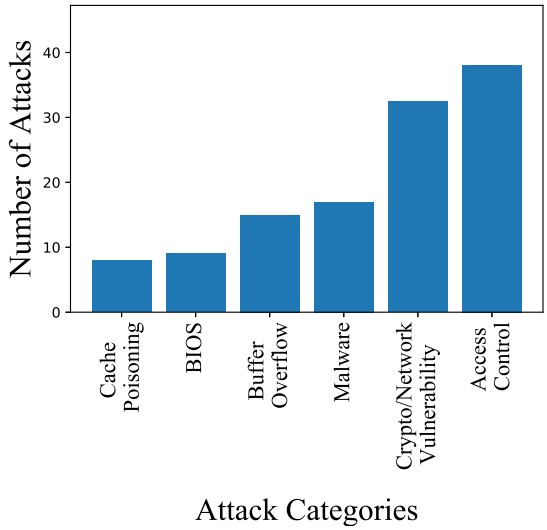


Fig. 7: Histogram depicting the number of novel exploits discovered in each category

Training accuracy refers to the accuracy of the SVM model when evaluated on the training dataset. Only four of the 140 training datapoints were incorrectly classified by the SVM model, yielding an accuracy of 97.2%. The test accuracy is manually determined by evaluating the feasibility of all the 1192 possible branches in the attack DAG. We observed that 31 of the 153 positive predictions were incorrect. On the other hand, all the negative predictions were accurate. Thus, 1161 of the 1192 datapoints of the test dataset were classified correctly by the SVM model, yielding a test accuracy of 97.4%.

4.3.1 *Constraint Satisfaction Problem (CSP) Formulation*

In this section, we use a CSP formulation to analyze the feasibility of DAG branches. Constraint-based reasoning requires the creation of a set of constraints over the features of the branches, such that any branch satisfying all the constraints is deemed to be a feasible branch. CSP is widely used in program analysis techniques. Unlike traditional CSP-addressable problems, there are no deterministic rules governing the feasibility of a branch in our attack DAG. Thus, generating mathematical constraints that define the feasibility of a branch is not easy.

We use heuristics derived from the statistics of the training set to generate constraints. The constraints used to detect infeasible branches are inspired by the thresholds

derived to construct the negative examples in Section 4.2.3. The variables and symbols are defined next:

- Hamming distance - hd (integer value)
- Height difference of nodes - ht_{diff} (float value)
- Head-leaf connection - hl (Boolean value)
- Leaf-leaf connection - ll (Boolean value)

Experimentally, the following results were observed on the training set.

1) Mean Hamming distance between feature vectors of nodes of feasible branches = 2.93.
2) Mean Hamming distance between feature vectors of nodes of infeasible branches = 4.30.
3) Height difference between nodes of feasible branches: Minimum = -0.08; Mean = 0.997; Maximum = 2.
4) Height difference between nodes of infeasible branches: Minimum = -3.33; Mean = 0.071; Maximum = 3.33.
5) (Number of infeasible head-leaf or leaf-leaf branches/ Number of feasible head-leaf or leaf-leaf branches) = 4.

These observations are used to generate the following heuristic constraints. A branch is deemed to be infeasible if:

1) $ht_{diff} \leq -0.09$ or $ht_{diff} > 2$
2) $hd > 5$
3) if $(4 \leq hd \leq 5)$ and $((hl == \text{True}) or (ll == \text{True}))$

When the above constraints are applied to the test dataset, they yield 60 FNs. Comparing this to the results in Fig. 6b, we observe that CSP analysis performs better than many SVM models. However, the best SVM model yields zero FNs, thus performing significantly better than CSP. Since our primary objective is to minimize the number of FNs, we prefer SVM over CSP.

4.4 Discovering Unexploited Attack Vectors

An unexploited attack vector is one that exists in the DAG but has not yet been exploited in any documented real-world attack on IoT/CPS. This is unusual because the DAG was constructed from real-world attacks on IoT/CPS. The unexploited attack vectors embedded in the DAG can be discovered through linear search on the DAG. Every path from a head node to a leaf node corresponds to a unique attack vector. The attack DAG has 51 such paths. However, only 41 known attack vectors were considered while constructing the attack DAG. Thus, 10 unexploited attack vectors are obtained through a linear search of all the attack paths, a subset of which is shown in Fig. 8. These 10 unexploited attack vectors represent 10 additional ways to compromise an IoT/CPS.

New attack vectors emerge due to the convergence of multiple attack paths at common basic block(s). Such an occurrence is illustrated in Fig. 8. Fig. 8a and Fig. 8b represent two subgraphs of the attack DAG in Fig. 4. Fig. 8c shows the graph obtained by combining Fig. 8a and Fig. 8b at the common node titled "*Access ports of network*." Fig. 8d depicts the new paths obtained from the combination of the two graphs. The five new paths thus discovered correspond to five attack vectors that have not yet been exploited in real-world CPS/IoT attacks.

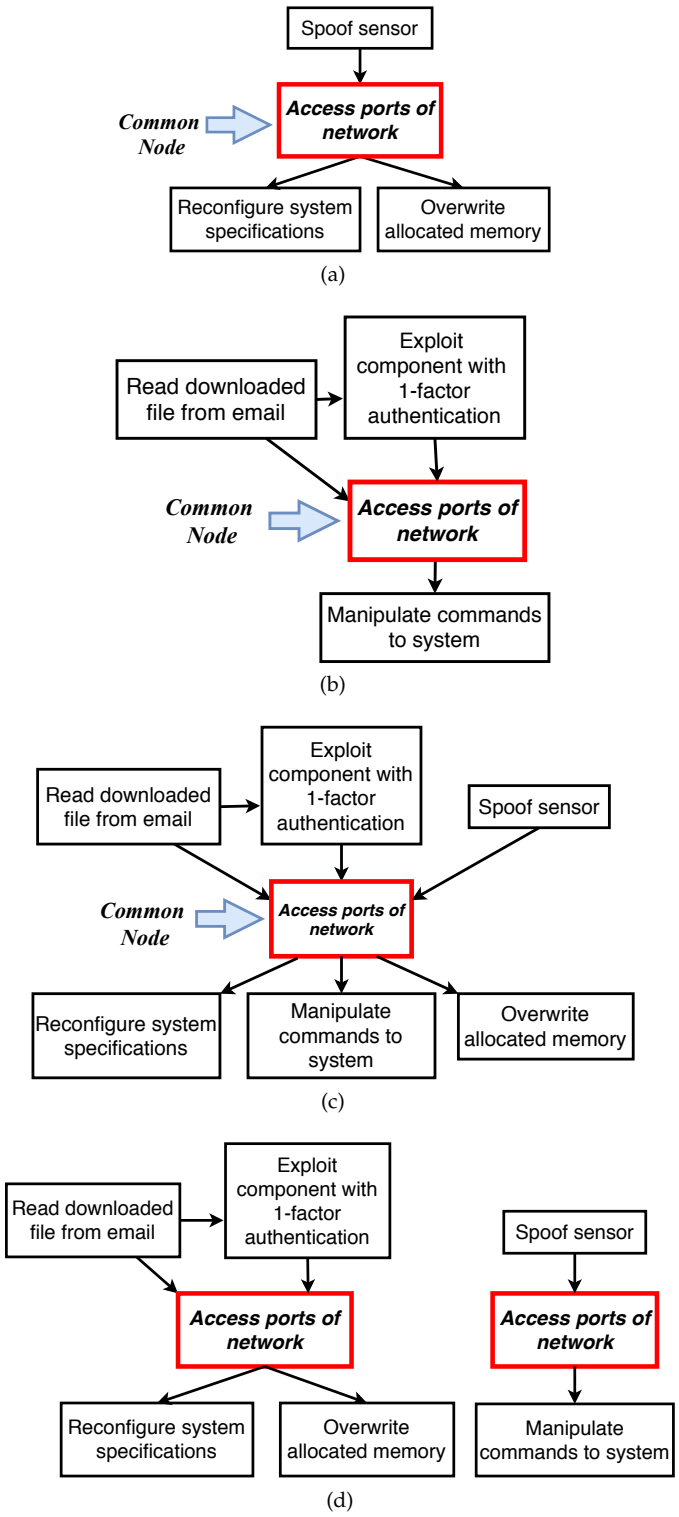


Fig. 8: Generating new exploits with linear search: (a) CDFG for $Attack_1$, (b) CDFG for $Attack_2$, (c) combined CDFG of both attacks, and (d) new attacks that emerge from a combination of the two CDFGs

5 IoT Case Study: Connected Car

The connected car is a complicated IoT system comprising various sensors, electronic control units (ECUs), system buses, and embedded software packages. It possesses a vast range of capabilities that includes Internet access, communication with multiple devices, and collection of real-time data

from the surroundings. While these functionalities enhance user convenience, they also expand the attack surface of the system. The attack surface of a connected car includes the networks of vehicle-to-vehicle communication, in-vehicular communication, and exposed software and sensors, to name a few [55]. We analyze the security of in-vehicular networks with the SHARKS framework. The most common entry points into the in-vehicular network are the ECUs, on-board diagnostics (OBD) port, WiFi, and GSM and bluetooth networks of the vehicle. Some of these are shown in Fig. 9.

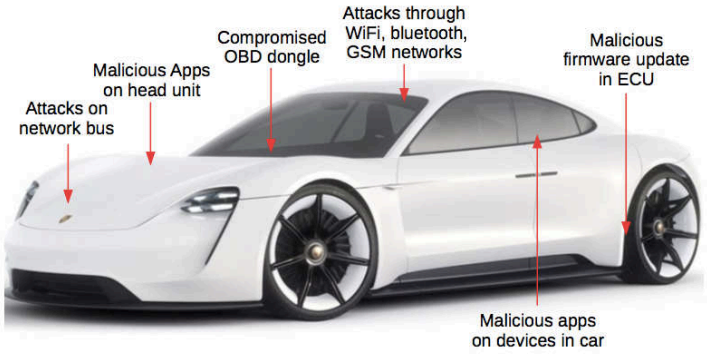


Fig. 9: Examples of hacker entry points into the connected car

The connected car has numerous ECUs that are responsible for different functionalities like anti-lock braking, lane departure warning, and engine management. All communications between ECUs occur over the network bus that connects all the ECUs to one another. There exist multiple networks for in-vehicle communications. Some of them are local interconnect network [56], FlexRay network [57], and media-oriented systems transport network [58]. One of the most popular in-vehicle networks is the Controller Area Network (CAN) [59]. CAN ensures real-time handling of all in-vehicle communications, including safety-critical data. This makes the security of the CAN bus critical to the safety and security of the smart vehicle. However, the CAN bus has been shown to be intrinsically not secure [60, 61]. Cryptographic techniques like encryption and message authentication cannot be applied to the data traversing the CAN bus. These operations increase the latency of processing the packets that leads to an increased ECU response time. This overhead is not permissible in the case of safety-critical, time-sensitive, and real-time applications. Cryptographic measures also prevent car mechanics from analyzing CAN traffic during troubleshooting. This is a major inconvenience for them because they generally use the CAN bus as a diagnostic tool during repair.

5.1 CAN Bus Vulnerabilities

Although CAN is the de-facto in-vehicle network in connected vehicles, it is not secure by design. The CAN protocol uses a broadcast mechanism for communication. Due to the absence of sender and receiver addresses in the data frames, every ECU can freely publish and receive messages from the bus. While this enables easier addition of new ECUs to the network, it poses a grave security threat to the system. We next discuss the popular vulnerabilities on the CAN bus that were detected by our approach.

1) **Frame sniffing:** The CAN protocol uses a broadcasting mechanism for ECU communications. This allows a malicious node on the CAN bus to receive all the data frames through sniffing. The absence of encryption makes it easier to analyze the collected frames. The range of valid messages on the CAN bus is small enough to be exhaustively analyzed. Fuzzing techniques can be used to decode the functionalities of various ECUs from the log of sniffed frames [62]. This is a breach of confidentiality of the system. Frame sniffing is often the precursor of more complex attacks.
2) **Frame spoofing:** Frame spoofing involves sniffing and reverse engineering of the data frames of the CAN bus. Using the details of the data frames, the adversary can broadcast malicious frames on the bus by spoofing a particular node. Absence of authentication schemes compromises the integrity of messages on the CAN bus. Spoofing attacks may result in incorrect speedometer readings, arbitrary acceleration of the vehicle, erroneous fuel level readings, and conveying malicious messages to the driver [63]. This poses grave safety concerns as the adversary can gain access to safety-critical ECUs like the braking system and engine management system.
3) **Denial of Service (DoS):** The CAN protocol implements a priority-based broadcasting communication scheme. For example, messages from the anti-lock braking system, which are critical to the safety of the passengers, are given higher priority for transmission on the bus than messages from climate control sensors. The priority of a frame is determined by a parameter id (PID). Lower values of PID signify higher priority messages. To launch a DoS attack, the adversary needs to decode the smallest acceptable value of PID from the history of CAN messages (obtained by frame sniffing). Then he can continually broadcast messages with the highest priority on the bus, thus preventing any other message from being transmitted on it [62]. This compromises the availability of the CAN bus to legitimate messages, thus denying service to these messages.
4) **Replay attack:** Replay attacks involve sniffing the frames on the CAN bus prior to launching the attack. Sniffing and analyzing the frame packets using fuzzing techniques reveal knowledge about the frame functionalities. Since the CAN protocol is bereft of authentication schemes and time-stamp verification, the recorded frame packets can be sent on the CAN bus at inconvenient time instances to launch various attacks. For example, the frame packet to unlock the car door can be replayed by a thief when the owner is not around. Replay attacks on cars have been demonstrated both in simulations [64] and real cars [62].

The other vulnerabilities that we consider in our experi-

ments are ECU buffer overflows [65] and malware injection through ECU firmware updates [66]. These attack vectors involve sending malicious packets to the ECUs over the CAN bus but do not involve exploiting any vulnerability of the CAN bus itself.

5.2 Application of SHARKS

In this section, we describe how we use our SHARKS approach to detect the aforementioned vulnerabilities in the given IoT system, namely the CAN bus. An adversary can gain access to the CAN bus through multiple entry points like the OBD port, WiFi, bluetooth, radio or the GPS system of the car [67]. In our simulations, we use the OBD-II port to gain access to the CAN bus. We simulate the CAN bus with OpenGarages ICSim simulator [67] on our workstation with LibSDL and Socket-CAN CAN-utils libraries. The simulation results can then be executed on a connected car, with the help of ScanTools software, by connecting the workstation (laptop) to the car through the OBD-II port.

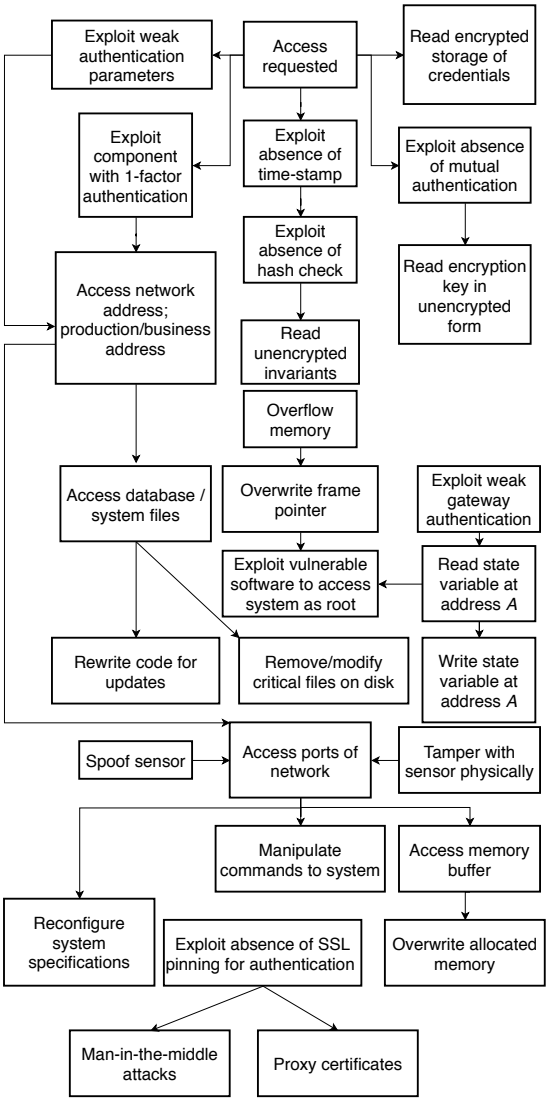


Fig. 10: The attack DAG for the CAN bus of the connected vehicle

To apply SHARKS to a specific CPS/IoT system, we have to design the attack DAG for it. The attack DAG shown in Fig. 4 is designed for a generic CPS/IoT system. The CAN bus has fewer functionalities than those considered during the design of the attack DAG in Fig. 4. This makes some of the nodes in the DAG in Fig. 4 redundant with respect to the CAN bus IoT system. We remove those nodes and obtain a subgraph of Fig. 4 that is relevant to the CAN bus. This subgraph, shown in Fig. 10, is referred to as the CAN attack DAG. It has 29 nodes, 27 branches, and represents 24 high-level attack vectors relevant to the CAN bus.

5.3 Results

We ran a pre-trained SVM model on the CAN attack DAG shown in Fig. 10. The SVM model was trained on the attack DAG in Fig. 4, and not on the CAN attack DAG. While testing the model's performance on the CAN attack DAG, we observed that it is able to discover 67 CAN vulnerability exploits that were initially absent in the CAN attack DAG. This indicates that our approach is generic enough to be deployed on any CPS/IoT system for vulnerability exploit detection. We classify the detected CAN bus vulnerabilities into the vulnerability categories mentioned in Section 5.1. Some of the attack branches predicted by the model and their corresponding categories are shown in Table 7.

TABLE 7: Novel exploits discovered

Branch discovered	**Vulnerability category**
Invariants unencrypted $\longrightarrow$ Read state variable at address A	Frame sniffing
No mutual authentication $\longrightarrow$ Man-in-the-middle attacks	Frame spoofing
No check for time-stamp $\longrightarrow$ Manipulate commands to system	Replay attack
Access memory buffer $\longrightarrow$ Write state variable at address A	ECU buffer overflow
Rewrite code for updates $\longrightarrow$ Download unwhitelisted software	Malware injection through ECU updates

The CAN attack DAG has 29 nodes and 27 branches. Putting $n = 29$ and $c = 27$ in Eq. (1), we observe that there are 785 datapoints in the test set. The SVM model predicts 88 of these to be feasible novel exploits and eliminates the rest. Manually examining the feasibility of the 88 positive predictions, we find that 67 of them are TPs. All the branches that were predicted to be negative are infeasible control/data flows. Hence, the SVM model reduced our search space from 785 to 88, which represents an 88.8% reduction in human effort. The confusion matrix of the predictions made by the model is shown in Table 8.

TABLE 8: Confusion matrix of SHARKS on CAN vulnerabilities

N=785	**Actual = No**	**Actual = Yes**	
Predicted = No	TN = 697	FN = 0	697
Predicted = Yes	FP = 21	TP = 67	88
	718	67	

6 Security Measures

In this section, the primary endeavor is to show how to defend CPS/IoT against the known attacks and novel exploits predicted by SHARKS at an optimal cost. Defense-in-depth and multi-level security (MLS) [68, 69] are the most appropriate schemes to adopt in such a scenario. Defense-in-depth refers to employing multiple defense strategies against a single weakness and is one of the seven properties of highly secure devices [70]. MLS categorizes data/resources into one of the following security levels: *Top Secret*, *Secret*, *Restricted*, and *Unclassified*. The first three levels have classified resources and require different levels of protection. Security measures become stricter as we move from Restricted to Top Secret. Many different policies can be employed to implement MLS in an organization. Some of the most popular policies are based on the Bell-La Padula (BLP) model [71] and the Biba model [72]. The BLP model prioritizes data confidentiality whereas the Biba model gives more importance to integrity.

The aggregated attack DAG is composed of multiple categories of attacks that are weaved together. Defense mechanisms can be systematically developed for each of these vulnerability categories in the form of defense DAGs. Defense DAGs mirror the corresponding attack subgraphs and make execution of the key basic blocks of the attack sequence infeasible. This ensures that no path from a head node to a leaf node in the attack DAG can be traversed in the presence of the suggested defense measures.

Many attacks have multiple defense strategies that can protect against them. The cost of our overall defense strategy increases with the complexity and number of defense measures that we enforce. Defense-in-depth helps us optimize this cost. The less sensitive resources (those belonging to the Restricted level) have basic defense measures against all attacks. As we move up the hierarchy to the Secret and Top Secret levels, we have more layers of security. Next, we demonstrate our defense strategies against access control and boot-stage attacks.

6.1 Defense against Access Control Attacks

Access control and privilege escalation attacks are the most common amongst real-world CPS/IoT attacks, as shown in Fig. 7. Access control attacks involve an unauthorized entity gaining access to a classified resource, thus compromising its confidentiality and/or integrity. Privilege escalation attacks involve an entity exploiting a vulnerability to gain elevated access to resources that it is not permitted to access. Implementation of strict policies can protect against such attacks. These security policies include multi-factor authentication, access control lists, and role-based access control. More layers of authentication, authorization, and network masking can be added for more sensitive resources. An example of a defense DAG is shown in Fig. 11.

In Fig. 11, we demonstrate the security measures deployed in a resource that is classified as Top Secret. A Top Secret resource is generally a computing device with more computing resources, the exploit of which can lead to a high-impact damage like the compromise of data confidentiality and integrity in a Top Secret data storage server. Hence, we should have multiple layers of authentication and network address masking, as shown in Fig. 11. In a resource classified as Restricted, there can be only one layer of authentication. More layers of authentication may be added for a Secret-level resource depending on the resources it has.

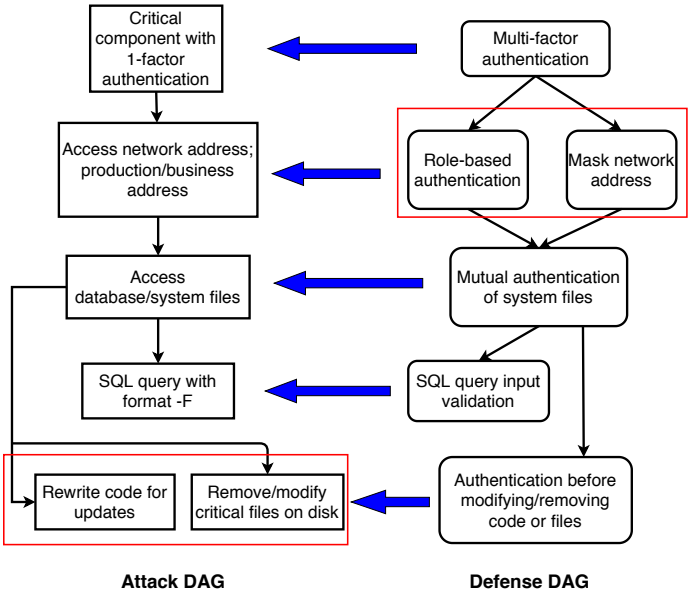


Fig. 11: Defense at the **Top Secret** level against access control and privilege escalation exploits. The DAG on the left depicts the attack DAG and the DAG on the right depicts the defense DAG. The arrows indicate the basic blocks of the defense DAG making the corresponding basic blocks of the attack DAG non-operational.

6.2 Defense against Boot-stage Attacks

This category of attacks is the most complicated among all the categories. While other attacks can be launched at the application level, these attacks typically require root access and have to be launched at the system level.

To defend against such attacks, a core root of trust for measurement is required along with a trusted platform module (TPM) or a hardware security module. These are generally present at a level lower than the kernel and sometimes referred to as the trusted computing base (TCB). In Fig. 12, the BOOTROM serves as the TCB. Defense against boot-stage attacks involves a series of hierarchical and chained hash checks of binary files and secret keys stored in the Platform Configuration Register (PCR) of the TPM. The PCR is inaccessible to all entities except the TPM. The detection of an incorrect hash value at any stage of the boot sequence causes the boot sequence to halt due to the detection of an illegal modification of the binary boot files and/or the secret(s). Fig. 12 gives an overview of the hash checks and execution of binary files at various levels.

7 Conclusion

The rapid advancement of CPS/IoT-enabling technologies, like 5G communication systems and ML, increases the scope of their applications manifold. Unfortunately, this also increases the attack surface of such systems that can often result in catastrophic effects. We have demonstrated

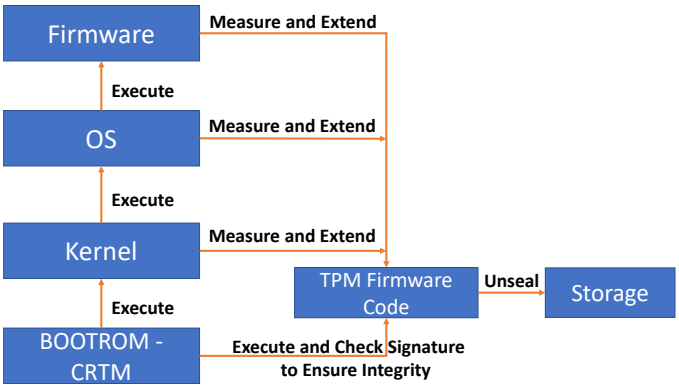


Fig. 12: Defensive measures against Boot-stage attacks

how ML can be used at the system and network levels to detect possible vulnerability exploits across the hardware, software, and network stacks of CPS/IoT. We discovered 10 unexploited attack vectors and 122 novel exploits using the proposed methodology and suggested appropriate defense measures to implement a tiered-security mechanism. We hope that this methodology will prove to be helpful for proactive threat detection and incident response in different types of CPS/IoT frameworks.

Acknowledgments

This work was supported by NSF under Grant No. CNS-1617628.

References

[1] H. Arasteh, V. Hosseinnezhad, V. Loia, A. Tommasetti, O. Troisi, M. Shafie-Khah, and P. Siano, "IoT-based smart cities: A survey," in *Proc. IEEE Int. Conf. Environment and Electrical Engineering*, 2016, pp. 1–6.

[2] A. O. Akmandor and N. K. Jha, "Smart health care: An edge-side computing perspective," *IEEE Consumer Electronics Magazine*, vol. 7, no. 1, pp. 29–37, 2018.

[3] M. Yun and B. Yuxin, "Research on the architecture and key technology of Internet of Things (IoT) applied on smart grid," in *Proc. IEEE Int. Conf. Advances in Energy Engineering*, 2010, pp. 69–72.

[4] S. K. Datta, R. P. F. Da Costa, J. Härri, and C. Bonnet, "Integrating connected vehicles in Internet of Things ecosystems: Challenges and solutions," in *Proc. IEEE Int. Symp. A World of Wireless, Mobile and Multimedia Networks*, 2016, pp. 1–6.

[5] X. Huang, P. Craig, H. Lin, and Z. Yan, "SecIoT: A security framework for the Internet of Things," *Security and Communication Networks*, vol. 9, no. 16, pp. 3083–3094, 2016.

[6] A. Mosenia and N. K. Jha, "A comprehensive study of security of Internet-of-Things," *IEEE Trans. Emerging Topics in Computing*, vol. 5, no. 4, pp. 586–602, 2017.

[7] T. Xu, J. B. Wendt, and M. Potkonjak, "Security of IoT systems: Design challenges and opportunities," in *Proc. IEEE/ACM Int. Conf. Computer-Aided Design*, 2014, pp. 417–423.

[8] M. Katagi and S. Moriai, "Lightweight cryptography for the Internet of Things," *Sony Corporation*, pp. 7–10, 2008.

[9] J. Lee, W. Lin, and Y. Huang, "A lightweight authentication protocol for Internet of Things," in *Proc. Int. Symp. Next-Generation Electronics*, 2014, pp. 1–2.

[10] G. E. Suh and S. Devadas, "Physical unclonable functions for device authentication and secret key generation," in *Proc. Design Automation Conference*, 2007, pp. 9–14.

[11] V. Sehwag and T. Saha, "TV-PUF: A fast lightweight analog physical unclonable function," in *Proc. IEEE Int. Symp. Nanoelectronic and Information Systems*, Dec. 2016, pp. 182–186.

[12] H. Suo, J. Wan, C. Zou, and J. Liu, "Security in the Internet of Things: A review," in *Proc. Int. Conf. Computer Science and Electronics Engineering*, vol. 3, Mar. 2012, pp. 648–651.

[13] A. O. Akmandor, H. Yin, and N. K. Jha, "Smart, secure, yet energy-efficient, Internet-of-Things sensors," *IEEE Trans. Multi-Scale Computing Systems*, vol. 4, no. 4, pp. 914–930, 2018.

[14] L. Xiao, X. Wan, X. Lu, Y. Zhang, and D. Wu, "IoT security techniques based on machine learning: How do IoT devices use AI to enhance security?" *IEEE Signal Processing Magazine*, vol. 35, no. 5, pp. 41–49, Sep. 2018.

[15] F. Copty, A. Kassis, S. Keidar-Barner, and D. Murik, "Deep ahead-of-threat virtual patching," in *Proc. Int. Wkshp. Information and Operational Technology Security Systems*, 2018, pp. 99–109.

[16] C. Zhang, J. Jiang, and M. Kamel, "Intrusion detection using hierarchical neural networks," *Pattern Recognition Letters*, vol. 26, no. 6, pp. 779–791, 2005.

[17] L. Sacramento, I. Medeiros, J. Bota, and M. Correia, "FlowHacker: Detecting unknown network attacks in big traffic data using network flows," in *Proc. IEEE Int. Conf. on Big Data Science And Engineering*, 2018, pp. 567–572.

[18] J. Cannady, "Artificial neural networks for misuse detection," in *Proc. Nat. Inf. Syst. Secur. Conf.*, 1998, pp. 443–456.

[19] A. Bivens, C. Palagiri, R. Smith, B. Szymanski, and M. Embrechts, "Network-based intrusion detection using neural networks," *Intelligent Engineering Systems through Artificial Neural Networks*, vol. 12, no. 1, pp. 579–584, 2002.

[20] C. Livadas, R. Walsh, D. Lapsley, and W. T. Strayer, "Using machine learning techniques to identify botnet traffic," in *Proc. IEEE Conf. Local Computer Networks*, 2006, pp. 967–974.

[21] S. Benferhat, T. Kenaza, and A. Mokhtari, "A naive Bayes approach for detecting coordinated attacks," in *Proc. Annual IEEE Int. Computer Software and Applications Conference*, 2008, pp. 704–709.

[22] G. R. Hendry and S. J. Yang, "Intrusion signature creation via clustering anomalies," in *Proc. Data Mining, Intrusion Detection, Information Assurance, and Data Networks Security*, vol. 6973, 2008, p. 69730C.

[23] M. Blowers and J. Williams, "Machine learning applied to cyber operations," in *Network Science and Cybersecurity*, 2014, pp. 155–175.

[24] K. Sequeira and M. Zaki, "Admit: Anomaly-based data mining for intrusions," in *Proc. ACM SIGKDD Int. Conf. Knowledge Discovery and Data Mining*, 2002, pp. 386–395.

[25] L. Bilge, E. Kirda, C. Kruegel, and M. Balduzzi, "EXPOSURE: Finding malicious domains using passive DNS analysis," in *Proc. Symp. Network and Distributed Systems Security*, 2011, pp. 1–17.

[26] C. Kruegel and T. Toth, "Using decision trees to improve signature-based intrusion detection," in *Proc. Int. Wkshp. Recent Advances in Intrusion Detection*, 2003, pp. 173–191.

[27] L. Bilge, D. Balzarotti, W. Robertson, E. Kirda, and C. Kruegel, "Disclosure: Detecting botnet command and control servers through large-scale netflow analysis," in *Proc. ACM Annual Computer Security Applications Conference*, 2012, pp. 129–138.

[28] F. Gharibian and A. A. Ghorbani, "Comparative study of supervised machine learning techniques for intrusion detection," in *Proc. IEEE Annual Conference on Communication Networks and Services Research*, 2007, pp. 350–358.

[29] A. Årnes, F. Valeur, G. Vigna, and R. A. Kemmerer, "Using hidden Markov models to evaluate the risks of intrusions," in *Proc. Int. Wkshp. on Recent Advances in Intrusion Detection*, 2006, pp. 145–164.

[30] F. Amiri, M. M. R. Yousefi, C. Lucas, A. Shakery, and N. Yazdani, "Mutual information-based feature selection for intrusion detection systems," *Journal of Network and Computer Applications*, vol. 34, no. 4, pp. 1184–1199, 2011.

[31] Y. Li, J. Xia, S. Zhang, J. Yan, X. Ai, and K. Dai, "An efficient intrusion detection system based on support vector machines and gradually feature removal method," *Expert Systems with Applications*, vol. 39, no. 1, pp. 424–430, 2012.

[32] H. Chen and L. Jiang, "GAN-based method for cyber-intrusion detection," *CoRR*, vol. abs/1904.02426, 2019. [Online]. Available: http://arxiv.org/abs/1904.02426

[33] Y. Meidan, M. Bohadana, Y. Mathov, Y. Mirsky, A. Shabtai, D. Breitenbacher, and Y. Elovici, "N-BaIoT: Network-based detection of IoT botnet attacks using deep autoencoders," *IEEE Pervasive Computing*, vol. 17, no. 3, pp. 12–22, 2018.

[34] N. Islam, S. Das, and Y. Chen, "On-device mobile phone security exploits machine learning," *IEEE Pervasive Computing*, Apr. 2017.

[35] J. Bickford, H. A. Lagar-Cavilla, A. Varshavsky, V. Ganapathy, and L. Iftode, "Security versus energy tradeoffs in host-based mobile malware detection," in *Proc. Int. Conf. on Mobile Systems, Applications, and Services*, 2011, pp. 225–238.

[36] H. Sayadi, H. M. Makrani, O. Randive, S. M. P.D., S. Rafatirad, and H. Homayoun, "Customized machine learning-based hardware-assisted malware detection in embedded devices," in *Proc. IEEE Int. Conf. on Trust, Security and Privacy in Computing and Communications*, 2018, pp. 1685–1688.

[37] V. Shandilya, C. B. Simmons, and S. Shiva, "Use of attack graphs in security systems," *Journal of Computer Networks and Communications*, 2014.

[38] S. Jha, O. Sheyner, and J. M. Wing, "Two formal analyses of attack graphs," in *Proc. IEEE Computer Security Foundations Wkshp.*, June 2002, pp. 49–63.

[39] M. U. Aksu, K. Bicakci, M. H. Dilek, A. M. Ozbayoglu, and E. I. Tatli, "Automated generation of attack graphs using NVD," in *Proc. ACM Conf. Data and Application Security and Privacy*, 2018, pp. 135–142.

[40] L. Lu, R. Safavi-Naini, M. Hagenbuchner, W. Susilo, J. Horton, S. L. Yong, and A. C. Tsoi, "Ranking attack graphs with graph neural networks," in *Proc. Int. Conf. Information Security Practice and Experience*, 2009, pp. 345–359.

[41] M. Yousefi, N. Mtetwa, Y. Zhang, and H. Tianfield, "A reinforcement learning approach for attack graph analysis," in *Proc. IEEE Int. Conf. Trust, Security and Privacy in Computing and Communications*, Aug. 2018, pp. 212–217.

[42] L. Szekeres, M. Payer, T. Wei, and D. Song, "SoK: Eternal war in memory," in *Proc. IEEE Symp. Security and Privacy*, 2013, pp. 48–62.

[43] Y. Gao, L. Chen, G. Shi, and F. Zhang, "A comprehensive detection of memory corruption vulnerabilities for C/C++ programs," in *Proc. IEEE Int. Symp. Parallel & Distributed Processing with Applications*, 2018, pp. 354–360.

[44] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *Proc. IEEE Symp. Security and Privacy*, 2019, pp. 1–19.

[45] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown," *arXiv preprint arXiv:1801.01207*, 2018.

[46] C. Trippel, D. Lustig, and M. Martonosi, "Checkmate: Automated synthesis of hardware exploits and security litmus tests," in *Proc. IEEE/ACM Int. Symp. Microarchitecture*, 2018, pp. 947–960.

[47] Z. Kohavi and N. K. Jha, *Switching and Finite Automata Theory, 3rd ed. Cambridge University Press*, 2009.

[48] C. Cortes and V. Vapnik, "Support-vector networks," *Machine Learning*, vol. 20, no. 3, pp. 273–297, 1995.

[49] R. Langner, "Stuxnet: Dissecting a cyberwarfare weapon," *IEEE Security & Privacy*, vol. 9, no. 3, pp. 49–51, 2011.

[50] A. Humayed, J. Lin, F. Li, and B. Luo, "Cyber-physical systems security: A survey," *IEEE Internet of Things Journal*, vol. 4, no. 6, pp. 1802–1831, Dec. 2017.

[51] N. Aaraj, A. Raghunathan, and N. K. Jha, "Dynamic binary instrumentation-based framework for malware defense," in *Proc. Int. Conf. Detection of Intrusions and Malware, and Vulnerability Assessment*, 2008, pp. 64–87.

[52] F. Ghasemi, A. Mehridehnavi, A. Perez-Garrido, and H. Perez-Sanchez, "Neural network and deep-learning algorithms used in QSAR studies: Merits and drawbacks," *Drug Discovery Today*, 2018.

[53] S. Hassantabar, Z. Wang, and N. K. Jha, "SCANN: Synthesis of compact and accurate neural networks," *arXiv preprint arXiv:1904.09090*, 2019.

[54] C. Chang and C. Lin, "LIBSVM: A library for support vector machines," *ACM Trans. Intelligent Systems and Technology*, vol. 2, pp. 27:1–27:27, 2011.

[55] A. Lima, F. Rocha, M. Völp, and P. Esteves-Veríssimo, "Towards safe and secure autonomous and cooperative vehicle ecosystems," in *Proc. ACM Wkshp. on Cyber-*

Physical Systems Security and Privacy, 2016, pp. 59–70.
[56] M. Ruff, "Evolution of local interconnect network (LIN) solutions," in *Proc. IEEE Vehicular Technology Conference*, vol. 5, 2003, pp. 3382–3389.
[57] R. Makowitz and C. Temple, "FlexRay: A communication network for automotive control systems," in *Proc. IEEE Int. Wkshp. Factory Communication Systems*, 2006, pp. 207–212.
[58] B. T. Fijalkowski, "Media oriented system transport (MOST) networking," in *Automotive Mechatronics: Operational and Practical Issues*, 2011, pp. 73–74.
[59] S. Tuohy, M. Glavin, C. Hughes, E. Jones, M. Trivedi, and L. Kilmartin, "Intra-vehicle networks: A review," *IEEE Trans. Intelligent Transportation Systems*, vol. 16, no. 2, pp. 534–545, 2014.
[60] O. Avatefipour, A. Hafeez, M. Tayyab, and H. Malik, "Linking received packet to the transmitter through physical-fingerprinting of controller area network," in *Proc. IEEE Wkshp. on Information Forensics and Security (WIFS)*, 2017, pp. 1–6.
[61] W. Choi, H. J. Jo, S. Woo, J. Y. Chun, J. Park, and D. H. Lee, "Identifying ECUs using inimitable characteristics of signals in controller area networks," *IEEE Trans. Vehicular Technology*, vol. 67, no. 6, pp. 4757–4770, 2018.
[62] K. Koscher, A. Czeskis, F. Roesner, S. Patel, T. Kohno, S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, and S. Savage, "Experimental security analysis of a modern automobile," in *Proc. IEEE Symp. Security and Privacy*, 2010, pp. 447–462.
[63] J. Liu, S. Zhang, W. Sun, and Y. Shi, "In-vehicle network attacks and countermeasures: Challenges and future directions," *IEEE Network*, vol. 31, no. 5, pp. 50–58, 2017.
[64] T. Hoppe and J. Dittman, "Sniffing/replay attacks on CAN buses: A simulated attack on the electric window lift classified using an adapted CERT taxonomy," in *Proc. Wkshp. Embedded Systems Security*, 2007, pp. 1–6.
[65] S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, S. Savage, K. Koscher, A. Czeskis, F. Roesner, and T. Kohno, "Comprehensive experimental analyses of automotive attack surfaces," in *Proc. USENIX Security*, 2011, pp. 1–16.
[66] D. K. Nilsson and U. E. Larson, "Conducting forensic investigations of cyber attacks on automobile in-vehicle networks," *Int. Journal of Digital Crime and Forensics*, vol. 1, no. 2, pp. 28–41, 2009.
[67] B. R. Payne, "Car hacking: Accessing and exploiting the CAN bus protocol," *Journal of Cybersecurity Education, Research and Practice*, vol. 2019, no. 1, p. 5, 2019.
[68] D. of Defense: Washington DC, "Security requirements for automatic data processing (ADP) systems," *DoD Directive 5200.28*, Dec. 1972.
[69] ——, "Techniques and procedures for implementing deactivating testing and evaluating secure resource-sharing ADP systems," *DoD 5200.28-M*, Jan. 1973.
[70] G. Hunt, G. Letey, and E. Nightingale, "The seven properties of highly secure devices," *Tech. Rep. MSR-TR-2017-16, Microsoft Research*, 2017.
[71] J. Rushby, "The Bell and La Padula security model," *Computer Science Laboratory, SRI International, Menlo Park, CA*, 1986.
[72] K. J. Biba, "Integrity considerations for secure computer systems," MITRE Corp., Bedford, MA, Tech. Rep., 1977.

Tanujay Saha Tanujay Saha is currently pursuing his Ph.D. degree at Princeton University, NJ, USA. He received his Master's Degree in Electrical Engineering from Princeton University and Bachelors in Technology in Electronics and Electrical Communications Engineering from Indian Institute of Technology, Kharagpur, India in 2017. He has held research positions in various organizations and institutes like Intel Corp., KU Leuven, and Indian Statistical Institute. His research interests lie at the intersection of IoT, cybersecurity, machine learning, embedded systems, and cryptography.

Najwa Aaraj Najwa Aaraj is a Chief Research Officer at the UAE Technology Innovation Institute. She holds a Ph.D. in Electrical Engineering from Princeton University and a Bachelor's in Computer and Communications Engineering from the American University in Beirut. Her expertise lies in applied cryptography, trusted platforms, secure embedded systems, software exploit detection/prevention, and biometrics. She has over 15 years of experience working in the United States, Australia, Middle East, Africa, and Asia with global firms. She has two patents and 15 academic publications. She has worked in a cybersecurity start-up (DarkMatter). Prior to joining DarkMatter, she worked at Booz & Company, where she led consulting engagements in the communication and technology industry for clients across four continents. She has also held research positions at IBM T. J. Watson Center, New York, Intel Security Research Group, Portland, Oregon, and NEC Laboratories, Princeton, New Jersey.

Neel Ajjarapu Neel Ajjarapu is currently pursuing his B.S.E in the Department of Electrical Engineering at Princeton University, with a concentration in security and privacy as well as a certificate in technology and society from the Center for Information Technology Policy and Keller Center for Entrepreneurship. He has held intern positions at Microsoft Corp. and One Million Metrics Corp. (Kinetic) in product management and hardware engineering. His current research interests focus on automotive security, embedded systems, and cybersecurity.

Niraj K. Jha Niraj K. Jha received the B.Tech. degree in electronics and electrical communication engineering from I.I.T., Kharagpur, India, in 1981, and the Ph.D. degree in electrical engineering from the University of Illinois at Urbana-Champaign, Illinois, in 1985. He has been a faculty member of the Department of Electrical Engineering, Princeton University, since 1987. He was given the Distinguished Alumnus Award by I.I.T., Kharagpur. He has also received the Princeton Graduate Mentoring Award. He has served as the editor-in-chief of the IEEE Transactions on VLSI Systems and as an associate editor of several other journals. He has co-authored five books that are widely used. His research has won 20 best paper awards or nominations. His research interests include smart healthcare, cybersecurity, machine learning, and monolithic 3D IC design. He has given several keynote speeches in the area of nanoelectronic design/test and smart healthcare. He is a fellow of the IEEE and ACM.