A Machine Learning Based Monitoring Framework for Side-Channel Information Leaks

MICHAEL LESCISIN AND QUSAY H. MAHMOUD (Senior Member, IEEE)

Department of Electrical, Computer, and Software Engineering, University of Ontario Institute of Technology, Oshawa, ON L1G 0C5, Canada

CORRESPONDING AUTHOR: MICHAEL LESCISIN (e-mail: michael.lescisin@ontariotechu.net)

This work was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC).

ABSTRACT

Computer and network security is an ever important field of study as information processed by these systems is of ever increasing value. The state of research on direct attacks, such as exploiting memory safety or shell input errors is well established and a rich set of testing tools are available for these types of attacks. Machine-learning based intrusion detection systems are also available and are commonly deployed in production environments. What is missing, however, is the consideration of implicit information flows, or side-channels. Research has revealed side-channels formed by everything from CPU acoustic noise, to encrypted network traffic patterns, to computer monitor ambient light. Furthermore, no portable method exists for distributing side-channel test cases. This paper introduces a framework for adversary modeling and feedback generation on what the adversary may learn from the various side-channel information sources. The framework operates by monitoring two data streams; the first being the stream of side-channel cues, and the second being the stream of private system activity. These streams are used for training and evaluating a machine learning classifier to determine its performance of private system activity prediction. A prototype has been built to evaluate side-channel effects on four popular scenarios.

INDEX TERMS

Data mining, devops, machine learning, security, side-channel, software quality.

I. INTRODUCTION

Good software development practice teaches the fundamental rule that software security should be integrated into the complete development cycle of the software system and should not simply be an afterthought, a final step or a layer isolated from all other system concerns. The software development industry, as well as open source software communities, have acknowledged this design principle for many types of security vulnerabilities and found effective ways of integrating countermeasures against well-known types of software vulnerabilities. For example, when developing a REST based API that interacts with a SQL database it is well known that the API developer should thoroughly examine how data passed over the HTTP calls ends up in the SQL query strings in order to prevent a SQL injection attack. To aid in this process, tools are available such as sqlmap [1] which can automatically test code for SQL injection vulnerabilities.

Yet when it comes to the issue of side-channel security vulnerabilities, the same secure software design principle cannot be as easily applied. To a large extent, this is due to the fact that a side-channel vulnerability is generally more tailored to a specific threat model while more direct vulnerabilities should be present in all threat models. For example, if in any use case, a user is able to upload a string to a server which will cause a SQL database to be deleted, it is clear that a software vulnerability exists. Now consider the class of side-channel vulnerabilities. If an adversary is capable of learning, but not being able to modify, the commands executed on a remote server over SSH, it is not as clear-cut if a vulnerability exists. For certain security models, this may be acceptable as long as the arguments to these commands are not known. For other security models, this could be a requirements violation.

Another large reason for why side-channel vulnerability testing is difficult to integrate into the development process of
software is that the properties of a side-channel can often be
hardware dependent. For example, in [2], the authors describe
how by monitoring, via built-in computer microphone, the
subtle noises emitted by the power electronic components in
a computer monitor, an adversary capable of listening to the
microphone (such as a malicious Skype contact), could detect
basic information about the users screen activity. Considering
the large set of parties involved in the creation of this side-
channel (eg. monitor manufacturer, microphone manufacturer,
computer graphical interface designer), it is difficult to state
where this vulnerability originated - unlike the example of a
SQL injection where the origin of this vulnerability is the
failure to sanitize untrusted database inputs. In addition to
needling to test for this type of vulnerability on a wide variety
of hardware, thus increasing testing cost, there is no guarantee
about the side-channel properties in future hardware.

Due to the issues of complicated security requirements,
hardware-specific issues and unforeseen future use-cases, the
problem of verifying a software package for absence of side-
channel vulnerabilities continues to be a difficult task. For
this reason, we propose and implement a data-driven monitor-
ing framework, which monitors software system activity, ob-
servable hardware/software behaviours, and security model
properties. Based on these data sources, software quality
feedback is generated alerting developers, administrators, and
users of the status of security model violating side-channels.

The novelty of the work presented in this paper when com-
pared to the related work is that instead of focusing entirely
on a specific type of side-channel such as encrypted network
traffic generated by web applications or performance statistics
from the /proc file system, we present a data gathering
and analysis model which allows for side-channel data to
be gathered from multiple sources and allows for the poten-
tial demonstration of side-channels where leaked information
is recovered by selectively combining multiple side-channel
transmitter sources.

The source code for the data analysis performed by the
framework has been made available online [3]. The source
code for the instrumentation and data gathering utilities, with
the exception of the instrumentable virtual machine which
cannot be released at this moment due to a prior agreement
with a commercial research partner, is also available on-
line [3].

The rest of this paper is organized as follows: Section II
provides a review of the related work and discusses the ori-
gins and detection techniques for side-channels. Section III
presents the design of our framework while Section IV
presents the implementation. The framework is evaluated in
Section V while Section VI concludes the paper and presents
future work.

II. RELATED WORK

In the paper [4], Spreitzer et. al provide an overview of the
characterization and history of computer system side-channels
and thus have served to provide direction into the investigation
of the types of side-channels discussed in this paper. In terms
of a threat model, the authors categorize side-channel attacks
into three categories: local, vicinity, and remote.

Local side-channel attacks refer to attacks where the at-
tacker must have physical access to the device under attack.
For example, a side-channel attack that requires measurement
of the electrical potential of the device chassis would be con-
sidered to be a local side-channel attack.

Vicinity side-channel attacks refer to attacks where the
adversary is required to be in some physical proximity to
the device under attack. For example, in [5], the researchers
demonstrate how by monitoring the channel state information
(CSI) values of a Wi-Fi link, an adversary could gain insight
into which smartphone keys were typed as the position of the
users hand has an influence on the state of the Wi-Fi link. This
attack would be considered to be a vicinity side-channel attack
as the adversary must be within the range of the victim’s Wi-Fi
signal.

Remote side-channel attacks are the most severe type of
side-channel attack as the adversary is not constrained to a
physical distance from the system but rather is global. For
example, measuring response times from the public services
offered by a server could inadvertently leak private informa-
tion about the server.

The paper [4] also discusses the notion of a software-only
side-channel attack. In this type of attack, the adversary only
needs to be capable of executing software of a lower privi-
lege level on the target device. This type of attack exploits
hardware or software implementation artifacts for obtaining
private information which, as defined by the security model,
should not be accessible at this lower privilege level. Ex-
amples of software-only side-channels can be found in [2]
where the security domain of the microphone learns private
information from the security domain of the monitor, in [6]
where an Android application infers keystrokes through the
analysis of accelerometer and magnetometer data and in [7]
where operating system scheduler interactions are exploited
for learning information about the execution of privileged
processes.

Shared caches are also often exploitable through side-
channel attacks. Regardless of the type of shared cache, all
caches impose the same property on systems that use them
- that is, if a data resource is quick to be read, it must have
been accessed recently but if a data resource is slow to be
read, it must have not been accessed recently. This creates
what is referred to as a cache timing side-channel as it allows
an attacker who measures the response time for requested
resources to learn the patterns of resource use by a victim.
By indirectly measuring memory access patterns of a victim
process through memory cache response times, researchers
were able to learn 96.7% of the bits of a 2048-bit RSA key
used for a signature operation in GnuPG as demonstrated in [8].

Another computer system optimization, especially effec-
tive for data that must move across networks or be stored in
long-term storage is data compression. As data compression
strives to remove all redundant information from the input

140
data source [9], it is likely that each unique data input source contains different amounts of redundant information, and thus produces a compressed output size that is unique to the input data source. Therefore when compressed data is encrypted, the message size may provide insight into the message contents.

Many types of side-channels, regardless of their cause, become a serious threat to application security when the application communicates with a remote host over an untrusted network. As encryption strives to hide the contents of data transmitted over untrusted networks, it makes no attempt to hide the approximate sizes or timings of these events. Research has shown that by simply analyzing the patterns of network packets, and not their actual contents, a man-in-the-middle (MITM) adversary could be capable of learning; user responses to an online questionnaire [10], the mobile apps currently running on a device [11], or even passwords entered into a SSH remote shell [12].

As side-channels pose a serious threat to computer security and privacy, research has been directed to the development of side-channel detection methods and tools. The ongoing research challenge of this field of study is addressing the wide variety of implementation artifacts that create side-channels along with effectively determining the security requirements of an application under side-channel attack.

SideBuster [13] is a tool created by Zhang et. al. for detecting private information leaks through encrypted traffic patterns in web applications. SideBuster works by performing static taint analysis on web application source code to determine when private information is sent to the server and when information is transmitted to the server on a condition that is dependent upon the value of a private variable. SideBuster then generates test cases to cause these events that handle private information, while the associated network traffic is recorded. After the user-interaction/network traffic dataset has been built, the information loss is quantified by measuring the entropy loss into the set of user/web application interactions provided by the traffic pattern samples.

In [14], the authors examine the work done with SideBuster but instead of using static analysis for obtaining the points in the application at which client-server communication occurs, a dynamic analysis approach is instead used. In this approach, the authors employed the Crawljax web crawler to generate a state-machine model, including page state changes done by JavaScript, of the web application under test. After state-machine model generation, the application then proceeds on to leak quantification where the Fisher criterion is used to quantify side-channel severity - that is to measure the ratio of variance between classes to variance within classes. A side-channel free system will have a Fisher criterion equal to zero as either the variance between classes (numerator) is equal to zero as all actions generate the same network traffic pattern or the variance within classes (denominator) approaches infinity as all actions generate completely random traffic patterns.

Considering software-only side-channel attacks, ProcHarvester [15] exploits performance statistics from the /proc file system obtained by unprivileged processes on the Android mobile operating system to infer application loading, website loading, and keyboard gestures. Similar to ProcHarvester, SCAnDroid [16] gathers data from various unprivileged Android API calls to detect website launches, application launches, and Google Maps queries. Similar to the research discussed in our paper, both ProcHarvester and SCAnDroid use a machine-learning approach where data samples are fed to a dynamic time warping (DTW) algorithm which classifies (i.e. identifies the private event which caused the data features) them by finding their closest match within the training database. In addition to /proc file system data, and unprivileged API calls, side-channels may be formed by the methods exposed to unprivileged browser JavaScript. In [17], the authors perform a statistical analysis on these browser properties and demonstrate the detection of the underlying operating system, CPU architecture, privacy-enhancing browser extensions, and exact browser version.

### III. FRAMEWORK DESIGN

In order to facilitate the detection of side-channel information leaks in applications, we propose a layered set of functional components which allow for the creation of side-channel test scenarios.

When developing a side-channel attack scenario using the proposed framework, the layered architecture assists by facilitating the reuse of components along the attack workflow. For example, a framework component which gathers ambient light information to be later used to detect how successfully an adversary could learn which application is in use could be replaced with a component that gathers information on instantaneous power consumption for the same purpose of application use identification.

The presented side-channel detection framework must be installed at a point where both publicly observable and private information flows can be monitored (Fig. 1). In the evaluation and results section of this paper, the publicly observable side-channel information sources are encrypted network traffic patterns and CPU power consumption. Private system events examined in this paper include commands executed in remote...
Fig. 2. An example deployment of the framework for auditing network traffic for side-channel information leaks. Network traffic is labelled with private activity. These labels are used for model training by the framework and then discarded.

shells and keys typed into a remote desktop session. Therefore, the framework must be able to log both of these private (accessed through privileged operations) and public (accessed through the monitoring adversary observable signals) flows of information so that the ability of a potential side-channel adversary inferring private events using only public events may be measured. Fig. 2 shows a possible deployment where the framework is deployed to a gateway server which receives public Internet traffic labeled with private event labels and uses this information to train a model side-channel adversary. At the gateway server, the private labels are removed so that a real adversary would not gain an advantage in learning private system activity patterns.

Splitting the workflow components of a side-channel attack into generalizable and orthogonal concerns, the proposed framework allows for rapid design of side-channel attack models for the purpose of ensuring that software security model requirements are being met. Specifically, the concerns are split into: a data gathering layer where both private system events and public system events are logged, a feature extraction layer where captured data is filtered thus creating a representation of the private/public system behaviour that is well suited for training machine learning classifiers, a machine learning layer where classifiers are trained and evaluated on their ability to predict private events given observed public side-channel events, a threat modeling layer where the performance measurements from the evaluated machine learning classifiers are evaluated against the system threat model, and a reactive layer where an action is performed based on the result from the threat modeling layer.

The remainder of this section of the paper will describe each of the five layers of the framework as well as the implemented functional components associated with each layer (Fig. 3).

A. DATA GATHERING LAYER

The data gathering layer consists of the components which are necessary for gathering the raw data from a source that, under the system security requirements, is considered to be observable by an adversary. The data which is gathered must contain both the observable traits and the associated information which according to the system’s security requirements is intended to remain private.

1) BASH SHELL UDP TAGGER
The purpose of the Bash shell UDP tagger is to add UDP packets (traffic tags) to the set of captured packets to denote the beginnings and endings of traffic bursts that flow from server to client and are the result of the execution of a shell command. This shell provides the same interface as a regular UNIX shell but has the property that it labels network traffic, via UDP packets, denoting the beginning and end of each command execution.
2) FIREFOX ADDON UDP TAGGER
The purpose of the Firefox addon UDP tagger is similar to the purpose of the Bash shell UDP tagger except instead of tagging shell commands executed, website URLs are tagged. All that is required to gather data pairs of HTTPS traffic samples and website URLs is the installation of this addon in Firefox.

3) TRAFFIC TAGGER FOR TUNNELED VNC
As will be discussed later in this paper, the feature extraction layer provides a SSH Labeled Sequence Extractor component whose purpose is to monitor network traffic at the endpoints of SSH tunnels and isolate sections of the SSH encrypted data stream according to selected events from the plaintext data stream entering or leaving the SSH tunnel. The role therefore of this traffic tagger for tunneled VNC is to select relevant events from a plaintext TCP stream carrying Virtual Network Computing Remote Framebuffer Protocol (VNC RFB) data. Specifically, the events of key press and key release are of interest as they have the potential to manipulate the remote display in ways distinguishable through network traffic side-channels (Fig. 4).

4) INSTRUMENTABLE TESTBENCH VIRTUAL MACHINE
In order so that the effects of hardware based side-channels can be reliably simulated while requiring a minimal amount of simulation code to be written, an instrumentable testbench virtual machine is included in the data gathering layer of the proposed framework. This instrumentable virtual machine allows high-level scripts to be written to simulate user interaction as well as gather data from simulated hardware and potentially alter its operation. Ultimately, the high-level goal of this instrumented virtual machine is the same as the previously discussed data gathering tools that is, to generate a log file of private interactions and correlated observable system behaviours.

B. FEATURE EXTRACTION LAYER
The feature extraction layer consists of the preliminary data processing that is required to be performed on the raw captured data (eg. recorded network traffic) in order to generate feature vectors (eg. histogram of HTTP object sizes). Through the studying of the related work as well as understanding how network applications work from high-level to low-level, methods are implemented in this layer which generate the features, which have been shown to be highly applicable for side-channel detection, from the supplied raw data. The following subsections describe the modular functional components found at this layer.

1) UDP LABELED SEQUENCE EXTRACTOR
This functional component proves itself to be useful when handling raw streams of encrypted network traffic labeled by UDP packets such as that which is generated by the Bash shell UDP tagger or the Firefox addon UDP tagger. Simply put, this method reads through a capture of network traffic and produces pairs of the label of the private activity and the ordered list of TCP payload sizes which are transmitted over the network during the time period for which the private event occurs.

2) SSH LABELED SEQUENCE EXTRACTOR
This functional component is instrumental for creating scenarios which evaluate the security and privacy of tunneling plaintext TCP protocols over SSH. This method is designed to work in the situation where network traffic to be analyzed is captured at a host which serves as the entry point or exit point of an SSH tunnel (Fig. 5). Specifically, this method excepts the following parameters:
- The set of captured network traffic packets.
- The source of the plaintext traffic (IP, port)
- The destination of the plaintext traffic (ie. SSH tunnel entry point) (IP, port)
- The source of the encrypted traffic (ie. SSH tunnel entry point) (IP, port)
- The destination of the encrypted traffic (ie. SSH tunnel exit point) (IP, port)
- A reference to a function which will label the encrypted traffic stream based on the events detected in the plaintext traffic stream

This method returns a list of tuples in the form (e, p) where e is the event descriptor for a private event and p is the set of TCP packet sizes of the SSH encrypted traffic stream that ensues the occurrence of the private event.

3) NBURST FILTER
This functional component is instrumental for building scenarios which involve analyzing encrypted web traffic as when web objects (ie. HTML documents, JavaScripts, etc...) are downloaded over HTTP they typically generate continuous sequences of full length TCP payloads. This is a consequence
of the fact that the sizes of typical web objects greatly exceed that of the maximum data unit of a TCP datagram [10]. This method accepts a list of integer numbers representing the sizes of frames/packets/payloads, along with the target packet size, referred to as $N$. The method then returns the list of byte counts for each continuous run of $N$ byte entities in the input sequence plus the size of the entity trailing the $N$ byte run.

4) TIME DENSITY FILTER
Similar to the goal of segmenting a traffic stream based on packet size patterns as is done with NBurst filtering, time density filtering works by segmenting a traffic stream based on the time intervals between packets. This functional component accepts four parameters:

- The set of captured network traffic packets.
- A reference to a function that returns a Boolean value indicating if a given packet matches the criteria for consideration (i.e. correct source/destination address).
- The target time spacing between packets.
- The tolerance of the time spacing.

The ultimate goal of using packet timing density filtering is to create clusters of packets based on bursts of network traffic. This functionality is instrumental for building scenarios involving real-time applications as this clustering reveals the presence of these real-time events for which further features can be extracted (such as total bytes transferred) in an attempt to learn private information about the event.

C. MACHINE LEARNING LAYER
The core role of the machine learning layer is to determine the accuracy of predicting private information for an adversarial party following a given machine learning algorithm. The machine learning layer is fed with the extracted features from the preceding feature extraction layer along with the private event labels. From this layer is generated a probabilistic model for the prediction of all events in the captured event space. It is the role of the following, threat modeling layer, to determine if the success of a machine learning adversary violates the security model for the system or application.

1) BALANCED LABEL DATA SPLITTER
This functional component splits a set of feature vectors and its corresponding labels into two disjoint sets for which one is used for model training and the other for model testing. In order to train the machine learning model to recognize feature vectors as accurately as possible, this method tries to keep the same proportions of labels in both the training and testing sets.

2) ENTITY HISTOGRAM CREATOR
This functional component provides a means to map a list of feature vectors of varying dimensionality into a list of feature vectors of uniform dimensionality. This functional component builds a range set spanning the smallest seen value to the largest seen value, and transforms each input vector to a histogram of values with respect to the range set. This functional component proves itself to be useful when processing estimated object sizes downloaded in a HTTP session as the distribution of object sizes can often accurately distinguish one webpage from the others.

3) DECISION TREE BUILDER
This functional component receives a training dataset and testing dataset and constructs a set of decision tree trained classifiers from the training data. These classifiers are automatically evaluated based on the provided testing data and tuples are generated in the form $(T, P)$ where $T$ is the true value provided by the testing dataset and $P$ is the value predicted by the trained model. For this paper, we have used the scikit-learn default decision tree hyper-parameters, however, should these default values prove to be inadequate for certain use-cases, more optimized values could be calculated using a grid search technique. Additionally their are other trained classifier algorithms that could be employed for this task, but given that this paper is focused on demonstrating the feasibility of the proposed framework, we have selected the decision tree model as it provides satisfactory results. Selecting a more optimized classifier model would be outside the scope of this paper.

D. THREAT MODELING LAYER
The role of the threat modeling layer is to determine if the results from a trial run of the machine learning adversary are sufficient to violate the requirements set out by the system’s security model. It is here, at this layer, where the framework methods become protocol specific. Therefore, when a class of side-channel vulnerability is discovered, for example the ability to predict commands executed over SSH [12], a threat modeling method is created which, based on the success of the machine learning adversary for predicting the execution of commands, will determine if a side-channel information leak exists for a particular command.

1) SSH COMMAND PREDICTION EVALUATOR
This functional component accepts a machine learning model generated from the preceding machine learning layer and a shell command. Based on the evaluation run that is conducted during the construction of the machine learning adversary, this functional method returns the ratio of true positives to the total number of tests conducted corresponding to the given shell command.

2) HTTPS PAGE LOAD PREDICTION EVALUATOR
This functional component accepts a machine learning model generated from the preceding machine learning layer and a web url. Using the data from the evaluation run, the probability of successful adversarial prediction of a specific URL is returned.
3) VNC KEY PRESS PREDICTION EVALUATOR
Working in the same manner as SSH Command Prediction Evaluator and HTTPS Page Load Prediction Evaluator, this functional component returns the probability of successful adversarial prediction of a given key typed on the keyboard in a VNC session.

4) STRING ENTROPY CALCULATOR
The goal of this functional component is to measure the unpredictability of an authentication string (such as a password) requested by a system given its side-channels. To accomplish this, a machine learning model must be trained with authentication strings of known leading correctness and the corresponding traces of side-channel information.

The algorithm for estimating the unpredictability of the authentication string then proceeds as follows. The labels describing the percent leading correctness of a given trace are considered. For each label in the label set, there is a probability that the side-channel works properly; \( P(\text{True} | \text{Predicted}) \).
In order to calculate \( P(\text{True} | \text{Predicted}) \), Bayes’ rule \([18]\) is employed on the results obtained from the evaluation of the trained classifier. Thus, a list is obtained describing how effective the side-channel is at measuring the percent leading correctness of an entered string.

The next step in the algorithm is to find all permutations of working/non working side-channel detections and their associated probabilities.

The third step in this algorithm, now that the permutations and their associated probabilities are known, is to calculate the estimated number of guesses required to obtain the correct value of the authentication string. To do so, each section between working side-channel markers is considered and thus the amount of trials required to obtain the authentication string is the sum of powers for each substring bounded by the working side-channel markers.

This expected trials calculation is repeated for all permutations and their results are summed together thus obtaining the expected number of trials required for breaking the authentication string under the side-channel affected system. Lastly, this number of expected trials is converted to a corresponding information entropy by evaluating the base-2 logarithm function against it.

E. REACTIVE LAYER
The role of the reactive layer is, as implied by the name, to react to any differences between a system’s security model and the results of the execution of a scenario’s threat modeling layer. This layer closes the quality control feedback loop by providing the next step for the mitigation of side-channel information leaks.

1) WARNING LOGGER
This functional component accepts a warning message, a warning threshold and the evaluated threat model score and prints the warning message if threat model score exceeds warning threshold. The selection of the warning threshold value varies between attack scenarios but generally is inversely related to event detection severity, and directly related to attack difficulty.

2) POLAR GAUGE RENDERER
This functional component accepts a minimum value, a maximum value, a reading value, and an output filename. This functional component then renders a circular gauge image for the range between the minimum value and maximum value and draws the reading needle at the reading value. This rendered image is then written to the output file specified by filename. A popular type of goal for the images rendered by this functional component is the creation of software performance dashboards for continuous integration systems.

3) BAR GRAPH RENDERER
Similar to the Polar Gauge Renderer, this functional component accepts a minimum value, a maximum value, a set of categories (x-axis data), a set of values (y-axis data) and a filename. This functional component renders to the image specified by filename, a bar graph where the values are plotted with respect to the categories. Just like the Polar Gauge Renderer functional method, the images rendered by this method are also suitable for performance dashboards.

4) ENTROPY TARGET RENDERER
This functional component generates an image which visually represents the reduction of the search space when determining the value of a private variable using side-channel information. This is visualized as a two colour target where the reduced search space is painted in green over-top of the original search space (red). The area of each coloured region is proportional to the size of the search space, therefore, if the monitored side-channels reveal little to no private information, the infographic will be almost entirely green but if the monitored side-channels reveal effective hints into the value of a private variable, the generated info-graphic will be, in large part, red.

5) REPORT WEBPAGE GENERATOR
Report Webpage Generator is a functional component within the reactive layer that allows for the rendering of an HTML webpage based side-channel health dashboard. This component allows for the creation of infocards which contain a title and an image file to be displayed. For example, the title could be predictability of executing the command `ls /dev` and the image file could be a gauge image measuring this predictability. After all infocards have been created, this functional component renders these infocards as a HTML document. This HTML document, and its associated resources, can be served with a static webpage server thus providing a dashboard describing the security performance of a system.
IV. IMPLEMENTATION

The presented side-channel detection framework was implemented using several open-source tools and libraries. This section of the paper details these components as, for each component, a summary of its design is presented, along with its role in the presented framework, followed by the reasons for why it was chosen.

A. DOCKER

Docker is a framework which employs and abstracts capabilities in the Linux kernel to create containers which isolate processes from each other and from the regular host processes. In addition to providing isolation, Docker’s copy-on-write mechanism allows containers to be built based on other containers all while saving hard drive space as only the differences between the parent and child container need to be saved to disk [19].

The role which Docker plays in the implementation of the proposed framework is both the provision of Linux hosts, each performing the necessary roles for a side-channel attack scenario (client, server, adversary), as well as the packaging of the side-channel data analysis tools, specifically the feature extraction layer, machine learning layer, threat modeling layer, reactive layer, and depending upon the attack scenario, the data gathering layer.

Considering the framework requirement of simulating the network of a client, server, and adversary, Docker lends itself very well to this task [20]. Specifically, using Docker, a network interface is by default created which can be monitored thus capturing the network traffic that is exclusively associated with the container and not from other processes running on the same physical host.

B. SCAPY

Scapy is a powerful network packet generation, manipulation, and analysis tool which may be used independently through a read-evaluate-print-loop (REPL) shell or by a Python program through the form of a Python module [21].

As many of the types of side-channels evaluated in this paper are network traffic based side-channels, Scapy performs an important role on the feature extraction layer. Specifically, when a traffic stream is tagged with UDP packets denoting the beginnings and endings of events, Scapy is used along with a stateful parser to slice out the traffic packets in between the start and end tags thus building a dataset of traffic labels and traffic samples. Furthermore, as most of the network traffic which is analyzed in this paper is in encrypted form, the actual values of the bytes in the traffic stream are, in general, of little relevance. The more significant features of the traffic, when reading side-channel information are the sizes of the application layer payloads and the times of the packets. In the implementation of the proposed framework, Scapy is used for transforming a specific packet from an encrypted stream sequence into a more general representation of a payload size or an event time. Lastly, for detecting side-channels, the framework will have to, at times, analyze unencrypted protocols. This typically occurs at the entry points and exit points of SSH tunnels. Therefore, in the implementation of the proposed framework, Scapy is used to write analyzers of plaintext protocols to detect the presence of specific events so that the encrypted traffic stream carrying these events may be tagged with the appropriate event labels and later analyzed.

C. BOCHS

Bochs is an x86 PC emulator, with a minimalist design requiring no host hardware acceleration thus making it easily portable to different architectures and operating systems [22]. In the implementation of Bochs, every emulated computer functionality (eg. memory access, instruction execution, secondary storage access, etc.) is implemented by a C++ method thus facilitating emulated hardware modification. Additionally, as no machine language translation occurs, the exact amount of emulated CPU instructions can be counted, and therefore greater insight can be obtained on timing side-channel properties, without interference from CPU resources used by host processes.

In the proposed framework, Bochs is employed whenever the security model of an application requires that the application be free of some hardware based side-channel. Using the modified version of Bochs that is integrated with the implementation of the proposed framework, one may develop scenarios where observable events such as hard drive accesses are monitored and correlated with unobservable events such as specific memory location accesses, thus building a machine learning adversary model describing how well an adversary could predict these private events.

D. SCIKIT-LEARN

Scikit-learn is a Python module which provides interfaces for a wide variety of machine learning algorithms [23]. Data is passed to and from these machine learning algorithms using Numpy arrays [24] thus allowing for compatibility with many other Python modules as Numpy is a commonly used format among scientific computing modules for Python. The Scikit-learn module is designed strongly around the concept of object oriented interfaces thus simplifying the switch from one classification algorithm (eg. Decision Tree Learning) to another (eg. KNeighbors).

The role which Scikit-learn plays in the proposed framework is the provision of the classification machine learning algorithms used by the machine learning layer. The classification sub-discipline of machine learning is appropriately suited to the task of side-channel detection as the result of classifying captured publicly observable system behaviours, and comparing the predicted results to the true private events will indicate if a side-channel is present.

E. MATPLOTLIB

Matplotlib is a Python module which produces production quality mathematical figures while following the same interface design patterns as MATLAB [25]. Just like Scikit-learn, Matplotlib is fully interoperable with Numpy arrays.
The role which Matplotlib plays in the implementation of the proposed framework is the generation of visualizations in the reactive layer. For example, if it is required to generate a bar graph representing the probability of successful detection of a finite set of events, the reactive layer can employ Matplotlib to perform this task.

V. EVALUATION AND RESULTS
This section of the paper presents the evaluation of the layered side-channel detection framework. In this evaluation section, we demonstrate the complete stack working on four different attack scenarios - analysis of SSH console access traffic, analysis of HTTPS web browsing traffic, analysis of the VNC protocol tunnelled over SSH, and analysis of simulated power consumption of a password entry system. For all four side-channel attack scenarios, the presented framework is used for both dataset generation and dataset analysis. Detailed descriptions of the dataset generation procedures can be found in the proceeding subsections. The generated data and side-channel analysis code are available online [3].

A. ANALYSIS OF SSH CONSOLE ACCESS TRAFFIC
It is well known that the timing characteristics of information streams wrapped with SSH are not obfuscated. In [12], Song et. al explore the use of packet timing and sizing information to infer private information exchanged over a SSH session. In this evaluation, the work done by Song et. al is shown to still be relevant even twenty years after the publication of their paper. Specifically, in this evaluation, the data stream of SSH traffic for a remote user interacting with the Bash console of a Linux server is investigated.

In order to gather the required network traffic data and labels for the security analysis, the Bash Shell UDP Tagger is employed to add UDP labels to the beginnings and endings of Linux command outputs. The next step in this data processing pipeline is to convert this stream of labeled captured network traffic packets into a set of lists where each list in the set corresponds to the sizes of SSH stream packets sent from server to client for a given execution of a Linux command. An associated set provides the description of the Linux command associated with each traffic pattern. The UDP Labeled Sequence Extractor from the feature extraction layer is employed to produce these two associated sets.

After extracting the features, the next step in side-channel adversary simulation is to split the extracted features dataset into two distinct sets with one for adversary training and the other for adversary testing. In this evaluation example, 70% of the dataset was used for training while the remaining 30% was used for testing. The exact datasets used for training and testing are displayed on the console upon execution of this scenario if a detailed description of the data used in this experiment is desired from the reader.

The adversary in this example is simulated by a decision tree trained classifier and thus after building this machine learning adversary model a list of tuples of the form (truevalue, predictedvalue) is returned. In the threat modeling layer, the security requirements of the system are evaluated against the machine learning adversary model. During this evaluation, a security model has been defined with the requirement that the execution of the UNIX commands {pwd, ls, ls/dev} shall not be detectable by a wire-tapping adversary. To verify this requirement, the SSH Command Prediction Evaluator is applied to the output of the machine learning adversary model thus obtaining the probabilities of successful command prediction for each command.

Lastly, after the comparison of the security requirements with the machine learning adversary model, this proposed framework reacts by generating a web-based dashboard which, similarly to Figure 6, graphically describes the probability of successful UNIX command prediction by the simulated adversary. The Polar Gauge Renderer and Report Webpage Generator functional components are employed in the dashboard generation and evaluation results can be obtained by executing our source code example [3].

B. ANALYSIS OF HTTPS WEB BROWSING TRAFFIC
The design of the HTTPS protocol works solely to encrypt and authenticate the underlying HTTP traffic stream and the properties of hiding when traffic streams begin and end or the amount of data exchanged in a session are not requirements for this protocol. Research has shown that under many circumstances, being able to learn the byte size of an HTTP session is sufficient to identify the webpage that was loaded. Furthermore, if the web page that is loaded is a result of a private user interaction (eg. form submission) then the private user interaction can also be learned by the adversary [10]. In this evaluation, the merits of the work done in [10], [13], and [14] are demonstrated by evaluating the success level of a wire-tapping adversary inferring which Wikipedia pages were loaded over HTTPS.

A singular run of the test is as follows; the user loads the main page of Wikipedia at https://en.wikipedia.org/wiki/Main_Page. This evaluation was conducted on
September 17th 2018 when the main page contained a hyperlink to the Wikipedia article on Toronto [26]. In this article, the user then clicks on CN Tower and loads this article. Next, the user clicks the link for, and visits the article on First Canadian Place. Lastly, the user then follows the link for the article PATH. The URLs visited in this run are as follows:

- https://en.wikipedia.org/wiki/Main_Page
- https://en.wikipedia.org/wiki/Toronto
- https://en.wikipedia.org/wiki/CN_Tower
- https://en.wikipedia.org/wiki/First_Canadian.Place
- https://en.wikipedia.org/wiki/PATH_(Toronto)

This test is evaluated for a total of four runs.

In order to collect automatically labeled network traffic samples from the loading of these article pages, the framework Firefox addon was used to inject UDP packets on port 5005 carrying payloads marking when web page loads begin and when they end along with their associated URLs.

The first step of the feature extraction phase taking place in this example is identical to the example of predicting executed UNIX commands. The UDP Labeled Sequence Extractor method is employed to generate a dataset mapping URLs visited to HTTPS stream (port 443) packet sizes moving from server to client. The second phase of feature extraction consists of applying the NBurst Filter with N set to 1370 to recover the approximate sizes of HTTP objects downloaded in the session. At the end of these two phases, the data is now ready to be sent to the Machine Learning Layer.

The simulated attack scenario then proceeds with a transformation of the variable length HTTP object size lists to fixed dimensionality feature vectors through the use of the Machine Learning Layer’s Entity Histogram Creator. To improve accuracy, each uniform length histogram vector is augmented with the count of web objects downloaded in a session. These uniform length feature vectors are then used to train and evaluate a decision tree model using the Machine Learning Layer’s Decision Tree Builder.

The security model for this example is relatively simple - based on the traffic patterns generated from the loading of each of these five articles, an adversary should not be able to determine which one was loaded beyond the random guess probability of 20%. Therefore, the threat modeling layer in this example simply calculates the probability of an adversary correctly guessing which article was loaded.

Similar to the example where the execution of UNIX commands was predicted by a machine learning simulated wire-tapping adversary, the reactive layer for this example renders an HTML dashboard showing the probabilities of successful article URL prediction by a similar type of adversary (Fig. 6).

As is one of the intended goals of our framework, component reuse can be seen on layer 2 as the same functional component which is used in the SSH console example is used again to build the template dataset from the UDP labeled traffic stream. Software component reuse also occurs on layer 3 where the same Decision Tree Builder component that is used in the SSH console example is used once again in this scenario. Lastly, software component reuse occurs on layer 5 with the same methods as the SSH console example being invoked for the rendering of the performance dashboard.

C. ANALYSIS OF SSH TUNNELED VNC TRAFFIC

In this evaluation, the security of the popular practice of tunneling unencrypted Virtual Network Computing (VNC) traffic over the encrypted SSH protocol is examined. When designing this attack scenario it is important to keep in mind that SSH tunneling of TCP connections does not hide their timing or size properties. The design of the VNC protocol must also be considered. As the VNC protocol is designed to be bandwidth efficient, only the sections of the remote desktop screen which have changed are updated. While this protocol design is effective for minimizing required network bandwidth, it suffers from a major security flaw. Due to the fact that the region to be changed of a screen can vary greatly in size, the amount of network traffic generated strongly correlates with the amount of space updated on the screen. In this evaluation, the ability for an adversary to determine which character was typed into a text editor over SSH tunneled VNC is evaluated. To the best of our knowledge, no research has yet been conducted on this type of side-channel information leak and therefore the existence of this type of information leak as well as the demonstration of its detection using the presented framework can both be considered as research contributions.

The setup for this network security experiment is as follows. A Docker container representing the remote desktop runs the Xvnc and openssh servers as well as the Geany text editor so that a connecting client may type characters which appear on-screen. In addition, there is another Docker container which simply runs an openssh client. The complete setup is that from the host computer, a VNC client makes an unencrypted connection to this openssh client container. This connection is received by the openssh client and is sent over SSH tunnel to the Xvnc Docker container. The result is that by running tcpdump in the openssh client container, one may observe both the plaintext protocol data from the host as well as the SSH tunneled data as it moves to and from the remote desktop container.

In terms of test data, all twenty-six lowercase letters of the English alphabet were typed into the Geany text editor for six iterations. Splitting the data evenly into the first half for training and the second half for testing yields three examples of each lowercase letter in both the training and testing data sets. We believe that our set of input characters is sufficient as the amount of remote desktop pixels changed as a result of a key being typed is independent of previous keys typed and therefore character n-grams need not be considered.

As explained in all other side-channel detection examples, a set of labels mapping user interaction events to
adversary-observable network traffic patterns is required. Taking advantage of the availability of both the plaintext and SSH tunneled versions of the VNC communication, the role of the data gathering layer in this scenario is to, by parsing the plaintext VNC traffic, determine the times and types of the keypress and keyrelease events and extract the associated encrypted server-to-client network traffic.

In order to build the required dataset mapping GUI interaction to generated encrypted network traffic, the SSH Labeled Sequence Extractor method from the Feature Extraction Layer is employed. This generated dataset is then sent to the Machine Learning Layer.

The machine learning layer for this scenario is built in the same way as the scenarios for the SSH console side-channel and the HTTPS web browsing side-channel. The dataset of keypresses and associated encrypted network traffic patterns is split into training and testing sets and a decision tree classifier is evaluated to predict how successful an adversary would be at predicting which key was typed over an SSH encrypted VNC session.

The threat modeling layer for this scenario is simple. The generated machine learning adversary model is queried for all letters a through z for the probability of successfully predicting the character typed.

The Reactive Layer for this scenario simply uses the Bar Graph Renderer functional component to draw a bar graph describing the probability of successful keypress prediction by a wire-tapping adversary (Fig. 7).

As is the intended framework design goal, this scenario employs software component reuse. At layer 3, the same Decision Tree Builder functional component from the previous two example scenarios is used once again for the prediction of user key presses. Although the functionality at layer 5 differs in this example, as a bar graph is drawn instead of a circular gauge, the interface is identical thus facilitating the quick change of data visualization technique if such change is desired.

D. ANALYSIS OF CPU TIMING WHEN CHECKING PASSWORD

It is well known that a string comparison algorithm which compares two strings character-by-character and returns false at the first inequality of characters is vulnerable to a timing side-channel attack as a greater equality of leading characters yields longer execution times. In the field of embedded systems, research has shown that power consumption pattern side-channels can be exploited to expose the section of code which the CPU is executing [27]. This technique becomes very effective when a CPU executes a HALT instruction as the purpose of this instruction is to place the CPU into a low power mode where no instructions are executed until it is brought back into operating mode by an interrupt. Processor power consumption therefore can be divided into two modes, operational and idle, where the transition from operational to idle occurs on a HALT instruction and the transition from idle to operational occurs on an interrupt. Through the measurement of time in the operational modes, an adversary may be able to determine which code sections were executed.

Considering the timing side-channel associated with the above discussed string comparison algorithm, this evaluation example will conduct a simulation of this algorithm running on an electronic door lock based on a minimal Linux based operating system. The system is designed so that in order to unlock the door, a 1000 character long string must be entered - such as from the scanning of a barcode. Using simulated power side-channel analysis, this evaluation example estimates the true information entropy of the required authentication string given the information provided to the adversary through the power side-channel.

The idea of using simulation to gather side-channel information traces from physical sources such as CPU electromagnetic emanations or power consumption is relatively new. The Rainbow [28] tool is an example of a modern tool based on QEMU’s emulation engine which generates information leakage model traces based on the simulation of code execution. Although the discovery of code execution time inference through CPU power monitoring is well known and cannot be considered as a research contribution of this paper, the successful entropy reduction of a private data string in a simulated computer system demonstrates the presented framework’s merit in the design process of secure systems. Simulation tools such as the Bochs-based one presented in this paper, and the QEMU-based Rainbow [28] demonstrate how side-channels can be detected and prevented in a cost effective manner by the use of simulation.

In order to gather the data describing the amount of instructions executed between idle modes, the Instrumentable Testbench Virtual Machine is employed and is configured with a controllable option which, when enabled, at each HALT instruction, the amount of CPU instructions executed since the previous HALT instruction is logged to an internal list data structure. When the state of the controllable option switches from on to off, the maximum value of the filtered set of

FIG. 7. The probability of success of a wire-tapping adversary predicting keys typed based on observing encrypted network traffic features.
When the minimal Linux operating system is idle, lengths of attack execution time data feature Linux system activity (Fig. 8, Entropy Target Renderer it is deployed into the threat modeling layer chosen as they follow the same design pattern and use side-channel information leaks filtered functional component, yet its interface is common to all other data visualization components described in this paper. Therefore, this Entropy Target Renderer component could easily be changed to a Polar Gauge Renderer if desired. Lastly, although the data in this example was gathered from an instrumented virtual machine, if the data were to be gathered from server response times, the UDP Labeled Sequence Extractor from layer 2 could be deployed once again making a total of three uses of the same component throughout this paper.

VI. CONCLUSION AND FUTURE WORK

In this paper, we have discussed the problem of side-channels in software systems and have proposed, as a core research contribution, a framework for side-channel detection. The evaluation of this framework has shown that it is indeed effective at detecting critical side-channel information leaks from common software system configurations, thus providing an additional research contribution of exposing the security flaws in these often thought to be secure configurations.

For future work, we would like to investigate alternative machine learning algorithms such as deep learning models and Support Vector Machines in place of the decision tree classifier and employ various performance metrics, such as F1-measure, for determining the best performing classifier models. These machine learning models would be evaluated both under our data generated for this paper, new data generated through the further exploration of the aforementioned data gathering experiments as well as public datasets. Additionally, we envision building datasets of interactions with applications and the side-channel cues that provide insight into these interactions [29] from the further investigation of the experiments conducted in this paper. Using these datasets, we envision the ability for a software developer to model their software system and have it automatically tested for side-channel information leaks before it is deployed into production.

REFERENCES

[1] M. Stampar, “Sqlmap - under the hood,” in PHDays 2013, Moscow, May 2013. [Online]. Available: https://www.slideshare.net/stamparm/ph-days-2013miroslavstampsqmapunderthehood

[2] D. Genkin, M. Pattani, R. Schuster, and E. Tromer, “Synesthesia: Detecting screen content via remote acoustic side channels,” in Proc. IEEE Symp. Secur. Privacy, San Francisco, CA, USA, 2019, pp. 853–869, doi: 10.1109/SP.2019.00074.
[3] M. Lescisin, “Sidechannel-detection-framework,” Jun. 2019. [Online]. Available: https://github.com/IntegralProgrammer/Sidechannel-Detection-Framework

[4] R. Spreitzer, V. Moonsamy, T. Korak, and S. Mangard, “Systematic classification of side-channel attacks: A case study for mobile devices,” IEEE Commun. Surveys Tut., vol. 20, no. 1, pp. 465–488, Jan.–Mar. 2018.

[5] M. Li et al., “When CSI meets public WiFi: Inferring your mobile phone password via WiFi signals,” in Proc. ACM SIGSAC Conf. Comput. Commun. Secur., ser. CCS’16, New York, NY, USA: ACM, 2016, pp. 1068–1079.

[6] A. R. Javed, M. O. Beg, M. Asim, T. Baker, and A. H. Al-Bayatti, “Alphalogger: Detecting motion-based side-channel attack using smartphone keystrokes,” J. Ambient Intell. Humanized Comput., pp. 1–14, 2020.

[7] A. Tannous, J. Trostle, M. Hassan, S. E. McLaughlin, and T. Jaeger, “New side channels targeted at passwords,” in Proc. Annu. Comput. Secur. Appl. Conf., Dec. 2008, pp. 45–54.

[8] Y. Yarom and K. Falkner, “Flush reload: A high resolution, low noise, 13 cache side-channel attack,” in Proc. 23rd USENIX Secur. Symp., San Diego, CA: USENIX Association, 2014, pp. 719–732.

[9] G. E. Blelloch, “Introduction to data compression,” 2013. [Online]. Available: https://www.cs.cmu.edu/~guyb/realworld/compression.pdf

[10] M. Lescisin and Q. Mahmoud, “Tools for active and passive network side-channel detection for web applications,” in Proc. 12th USENIX Workshop Offensive Technol, Baltimore, MD: USENIX Association, 2018. [Online]. Available: https://www.usenix.org/conference/woot18/presentation/lescisin

[11] S. Zhao, S. Chen, Y. Sun, Z. Cai, and J. Su, “Identifying known and unknown mobile network traffic using a multilevel classifier,” Secur. Commun. Netw., vol. 2019, pp. 9595081:1–9595081:11, 2019.

[12] D. X. Song, D. Wagner, and X. Tian, “Timing analysis of keystrokes and timing attacks on Ssh,” in Proc. 10th Conf. USENIX Secur. Symp. - vol. 10, ser. SSYM’01, Berkeley, CA, USA: USENIX Association, 2001. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251327.1251352

[13] K. Zhang, Z. Li, R. Wang, X. Wang, and S. Chen, “Sidebuster: Automated detection and quantification of side-channel leaks in web application development,” in Proc. 17th ACM Conf. Comput. Commun. Secur., ser. CCS’10, New York, NY, USA: ACM, 2010, pp. 595–606.

[14] P. Chapman and D. Evans, “Automated black-box detection of side-channel vulnerabilities in web applications,” in Proc. 18th ACM Conf. Comput. Commun. Secur., ser. CCS’11, New York, NY, USA: ACM, 2011, pp. 263–274.

[15] R. Spreitzer, F. Kirchengast, D. Gruss, and S. Mangard, “Procharchester: Fully automated analysis of proofs side-channel leaks on android,” in Proc. Asia Conf. Comput. Commun. Secur., ser. ASIACCS ’18, New York, NY, USA: ACM, 2018, pp. 749–763.

[16] R. Spreitzer, G. Palfinger, and S. Mangard, “Scandroid: Automated side-channel analysis of android apis,” in Proc. 11th ACM Conf. Secur. Privacy Wireless Mobile Netw., ser. WiSec ’18, New York, NY, USA: ACM, 2018, pp. 224–235.

[17] M. Schwarz, F. Lackner, and D. Gruss, “Javascript template attacks: Automatically inferring host information for targeted exploits,” in Proc. 26th Annu. Netw. Distrib. Syst. Secur. Symp., NDSS, San Diego, California, USA, Feb. 2019, pp. 24–27.

[18] M. F. Triola, “Bayes’ theorem,” in Elementary Statistics, 11th ed. Boston, MA, USA: Addison-Wesley, Jan. 2009.

[19] D. Merkel, “Docker: Lightweight linux containers for consistent development and deployment,” Linux J., vol. 2014, no. 239, pp. 76–91, Mar. 2014.

[20] J. Schipp, J. Dopheide, and A. Slagell, “Islet: An isolated, scalable, & lightweight environment for training,” in Proc. XSEDE Conf.: Sci. Advancements Enabled Enhanced Cyberinfrastructure, ser. XSEDE ’15, New York, NY, USA: ACM, 2015, pp. 17:1–17:6

[21] P. Biondi, “Packet generation and network based attacks with scapy,” CanSecWest, 2005. [Online]. Available: http://www.secdave.org/confl/scapy_csw05.pdf

[22] K. P. Lawton, “Bochs: A portable pc emulator for unix/x,” Linux J., vol. 1996, no. 29es, Sep. 1996. [Online]. Available: https://dl.acm.org/citation.cfm?id=326350.326357

[23] F. Pedregosa et al., “Scikit-learn: Machine learning in python,” J. Mach. Learn. Res., vol. 12, pp. 2825–2830, Nov. 2011.

[24] S. van der Walt, S. C. Colbert, and G. Varoquaux, “The numpy array: A structure for efficient numerical computation,” Comput. Sci. Eng., vol. 13, no. 2, pp. 22–30, Mar. 2011.

[25] J. D. Hunter, “Matplotlib: A 2d graphics environment,” Comput. Sci. Eng., vol. 9, no. 3, pp. 90–95, 2007.

[26] Wikipedia Authors, “Wikipedia Main Page,” Sep. 2018. [Online]. Available: https://en.wikipedia.org/wiki/Main_Page

[27] S. Hao, D. Li, W. G. J. Halfond, and R. Govindan, “Estimating mobile application energy consumption using program analysis,” in Proc. Int. Conf. Softw. Eng., ser. ICSE ’13, Piscataway, NJ, USA: IEEE Press, 2013, pp. 92–101.

[28] M. S. Pedro, V. Servant, and C. Guillemet, “Side-channel assessment of open source hardware wallets,” IACR Cryptol. Eprint Arch., vol. 2019, Art. no. 401.

[29] R. Benadjila, E. Prouff, R. Strullu, E. Cagli, and C. Dumas, “Deep learning for side-channel analysis and introduction to ASCAD database,” J. Cryptographic Eng., vol. 10, no. 2, pp. 163–188, 2020.

MICHAEL LESCISIN received the Bachelor of Engineering (B.Eng) degree in electrical engineering at Ontario Tech University. He completed graduate studies at Ontario Tech, and received the Master of Applied Science (MASc) degree in electrical and computer engineering, researching the use of machine learning for improving computer and network security and reliability.

QUSAY H. MAHMOUD (Senior Member, IEEE) is currently a Professor of software engineering with the Department of Electrical, Computer, and Software Engineering, Ontario Tech University, Oshawa, ON, Canada. He was the Founding Chair of the Department, and recently he was the Associate Dean with the Faculty of Engineering and Applied Science, Ontario Tech University. His research interests include intelligent software systems and cybersecurity.