A Monitoring Framework for Side-Channel Information Leaks

Michael Lescisin
Department of Electrical, Computer, and Software Engineering
Ontario Tech University
Oshawa, Ontario, Canada
Email: michael.lescisin@ontariotechu.net

Qusay H. Mahmoud
Department of Electrical, Computer, and Software Engineering
Ontario Tech University
Oshawa, Ontario, Canada
Email: qusay.mahmoud@uoit.ca

***Abstract*—Security and privacy in computer systems has always been an important aspect of computer engineering and will continue to grow in importance as computer systems become entrusted to handle an ever increasing amount of *sensitive* information. *Classical* exploitation techniques such as memory corruption or shell command injection have been well researched and thus there exists known design patterns to avoid and penetration testing tools for testing the robustness of programs against these types of attacks. When it comes to the notion of program security requirements being violated through *indirect* means referred to as *side-channels*, testing frameworks of quality comparable to popular memory safety or command injection tools are not available. Recent computer security research has shown that *private* information may be indirectly leaked through *side-channels* such as *patterns of encrypted network traffic*, *CPU and motherboard noise*, and *monitor ambient light*. This paper presents the design and evaluation of a *side-channel* detection and exploitation framework that follows a machine learning based *plugin* oriented architecture thus allowing *side-channel* research to be conducted on a wide-variety of *side-channel* sources.**

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 have 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 user's 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 needing to test for this type of vulnerability on a wide variety of hardware [3], 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 *monitoring framework*, which monitors *software system activity*, *observable 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*. A proof-of-concept implementation demonstrating the examples presented in the section *Evaluation and Results* is freely available on GitHub [4].

The rest of this paper is organized as follows. Section II presents a detailed analysis of the related work. The motivation for the detection of side-channel information leaks is discussed in Section III. The design of our framework is presented in Section IV. Section V discusses the results of our evaluation. Finally, concluding remarks and ideas for future work are presented in Section VI.

II. Related Work

In the paper [5], 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 attacker 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 considered to be a *local* side-channel attack. In [6], a *local* side-channel attack is demonstrated as *machine learning* classifiers are used to infer which Android application is running based on the profile of electromagnetic energy radiating from a system-on-a-chip (SoC) development board.

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 [7], 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 user's 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* information about the server [8].

The paper [5] 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 privilege 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. An example of a *software-only* side-channel is discussed in [2] where the security domain of the microphone can learn private information from the security domain of the monitor. Exploiting operating system scheduler interactions [9] or cache timing measurements [10] for learning information about the execution of privileged processes are also examples of *software-only* side-channel attacks.

Many types of *side-channels*, be they the result of a hardware implementation artifact, a shared cache, or any other 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; determining *private* inputs to web applications [11] or even learning passwords entered into a SSH remote shell [12].

III. Motivation

The reviewed research on the current state of side-channels and their detection provides a great variety of motivating examples for considering these types of vulnerabilities when designing secure software systems. The research concerning *static* and *dynamic* analysis techniques used for checking programs for side-channel information leaks is something that should be considered and implemented by testers of secure software. Indeed, the current state-of-the-art of side-channel detection can catch many of the implementation issues that result in information leaked via side-channels. For a *common* type of side-channel, such as a cache-timing attack or analysis of encrypted web application traffic, there exist tools capable of detecting if observable activity patterns are capable of distinguishing between user/application behaviours. However, if an example of a *less common*, but potentially severe, type of side-channel is discovered, the security community must build a detection tool from the ground up. This unnecessary effort leads to a lack of available side-channel detection tools and thus side-channel information leaks remain prevalent in computer systems.

Lastly, the available side-channel detection solutions consider exclusively *machine* behaviour and omit the effects of *human* behaviour. For example, when detecting side-channels formed by the size of network traffic bursts generated when downloading web pages over HTTPS, the current state-of-the-art considers exclusively the *features* associated with the traffic bursts. In reality, there is a great amount of information available in the time gaps between traffic bursts. These time gaps, in the context of web traffic, can result from *human* behaviour such as how much time was spent on a webpage depending upon how *interesting* the content was or which links on a page a user is most likely to click. In the context of the Internet-of-Things (IoT), the time gaps between *smart home* events can provide great insight into the activity of the human target. This lack of ability to accurately emulate *human* behaviour is indicative of the current state of research which is limited to modeling common machine configurations such as caches and encrypted network tunnels.

IV. Our Solution

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*. Each *scenario*, generates a stream of *private* system events (ie. events that should not be detected

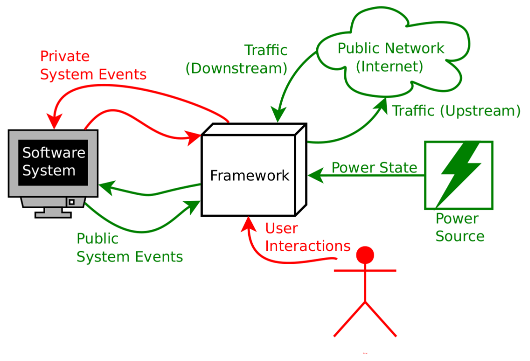


Fig. 1. The framework high-level architecture. Captures both public (green) and private (red) events.

by an adversary) and a stream of *public* system events (ie. events that are measurable by an adversary such as encrypted network traffic patterns). Based upon the success rate of a machine learning model predicting *private* events based on the *public* event stream, one can detect side-channels in their application and quantify their severity (Fig. 1).

The reason for choosing a *layered* architecture is for the promotion of component reuse. A popular example of a *layered* software architecture is the TCP/IP stack. When a developer wishes to create a new network service they only need to work on the appropriate layer of the stack. For example, if a developer is designing a RESTful API, they only need to write the HTTP server back-end as the methods for ensuring communication between this server and its clients are left to the lower levels of this stack thus *reusing* the software written by other developers. In a similar manner, when developing a side-channel attack *scenario* using the proposed framework, components along the attack workflow may be reused. 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 purpose of determining the success of a power analysis side-channel adversary on learning which application is in use.

Splitting the workflow components of a side-channel attack into *generalizeable* 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*.

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*. The format of this data stream is not specified and is likely to vary depending upon the nature of the data being captured. The only requirement for the formatting of the data sourced from the *data gathering layer* is that there is a module in the above *feature extraction layer* that is capable of extracting tuples of $(featurevector, label)$ from this stream.

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 in order to generate *feature vectors*. 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.

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.

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* at predicting the private event (eg. predicting the execution of a given command), will determine if a side-channel information leak exists.

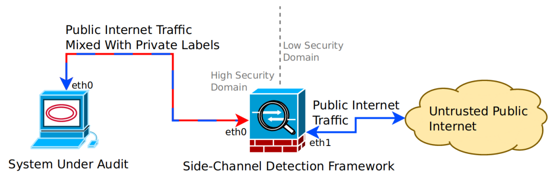
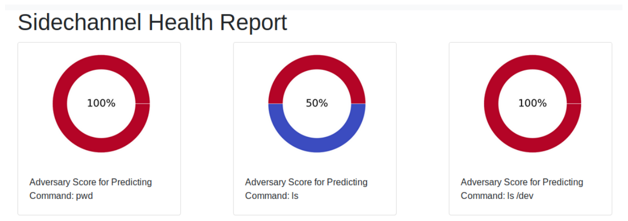


Fig. 2. An example deployment of the framework for auditing network traffic for side-channel information leaks. Network traffic is labeled with private activity. These labels are used for model training by the framework and then discarded.


Sidechannel Health Report

100%
Adversary Score for Predicting Command: pwd

50%
Adversary Score for Predicting Command: ls

100%
Adversary Score for Predicting Command: ls /dev

Fig. 3. The probability of success of a wire-tapping adversary predicting commands executed based on observing encrypted network traffic features.

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.

V. Evaluation and Results

In the following section of this paper, we present three scenarios involving side-channels formed by network traffic patterns. In each of these scenarios, traffic is labeled with *private* labels to serve as *training* data for the monitoring framework. In a production system deployment, these private labels are removed so as to not provide an advantage to any real adversaries (Fig. 2).

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 this evaluation, the data stream of SSH traffic for a *remote* user interacting with the *Bash* console of a Linux server is investigated. The exact datasets, containing the commands executed over remote shell, that were used for *training* and *testing* are available on GitHub [4] if a detailed description of the data used in this experiment is desired from the reader.

1) Data Gathering Layer: In order to train a model with the criteria necessary for determining what the *console user interaction* was, a set of *labels* describing the underlying user interaction associated with each traffic pattern is required. To generate these labels, UDP packets are sent from the *remote* server on port *5006* on the beginnings and endings of each *Bash* command executed thus associating the traffic pattern with a user activity.

2) Feature Extraction Layer: 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.

3) Machine Learning Layer: 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 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.

4) Threat Modeling Layer: 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.

5) Reactive Layer: Lastly, after the comparison of the *security requirements* with the *machine learning adversary model*, this proposed framework must react to the result of this comparison. In this evaluation, the *reaction* is to generate a web-based dashboard which graphically describes the probability of successful UNIX command prediction by the simulated adversary (Fig. 3).

From this example, it is shown that there are certain commands which can be detected when executed on a remote server over SSH. Practically speaking, one may want to hide the execution of these commands as they have the potential to reveal when sensitive operations are taking place thus making the server more vulnerable to targeted exploitation. This example provides a starting point for creating a plan for hiding these sensitive administration tasks done on a server.

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 [13]. In this evaluation, the success level of a wire-tapping adversary learning which *Wikipedia* page, out of a total five possible page loads, was loaded over HTTPS, is evaluated. Specifically, four captures of the data sequence were gathered on September 17th 2018 from the following pages:

- `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)`

1) Data Gathering Layer: In order to collect *automatically labeled* network traffic samples from the loading of these article pages, the framework Firefox addon from the *data gathering layer* 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.

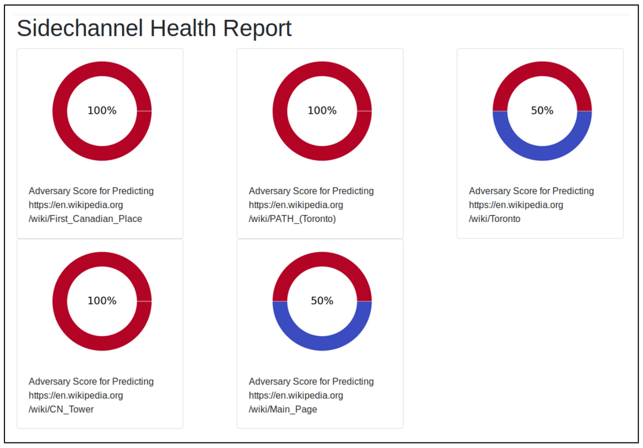
2) Feature Extraction Layer: The first step of the feature extraction phase taking place in this example is identical to the example of predicting executed UNIX commands; after parsing through the UDP labeled stream of HTTPS traffic, a dataset mapping URLs visited to HTTPS stream (port 443) packet sizes moving from server to client is generated. After simplifying the labels extracted from the labeled traffic stream, the second phase of *feature extraction* occurs. In this phase, for each list of packet sizes associated with the loading of a URL, the feature extraction layer is employed to transform this list of packet sizes to a list of approximate sizes of HTTP objects. This is done by generating a list of sums of TCP payload sizes for all continuous runs of *1370 bytes*.

3) Machine Learning Layer: The *machine learning layer* begins by splitting the generated dataset into 50% *training* and 50% *testing*. As the trained classifiers require the set of input data samples to be of uniform dimensionality, a transformation of HTTP object size lists of various lengths to a feature vector of *fixed dimensionality* is required. To accomplish this, an entity histogram (vector) is generated where each dimension is the count of objects of the given size. In order to further boost accuracy, each uniform length histogram vector is augmented with the *count* of web objects downloaded in a session. Once this *vector generation* process is completed, data vectors are split into *training* and *testing* subsets and are used to evaluate the performance of a machine learning classifier.

4) Threat Modeling Layer: 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.

5) Reactive Layer: 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. 4).

The results demonstrated in this subsection would be of interested to web application designers who would like to ensure that their clients' interactions with their application remain private. Using this provided example as a starting point, a web application security tester could group their application URL endpoints into *activity sets* and then use this framework to ensure that the different *activity sets* are not distinguishable from each other through network traffic pattern analysis.


Sidechannel Health Report

100%
Adversary Score for Predicting https://en.wikipedia.org /wiki/First_Canadian_Place

100%
Adversary Score for Predicting https://en.wikipedia.org /wiki/PATH_(Toronto)

50%
Adversary Score for Predicting https://en.wikipedia.org /wiki/Toronto

100%
Adversary Score for Predicting https://en.wikipedia.org /wiki/CN_Tower

50%
Adversary Score for Predicting https://en.wikipedia.org /wiki/Main_Page

Fig. 4. Similar to the UNIX command prediction example, this dashboard displays how well an adversary can predict which Wikipedia article was loaded.

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. To further improve bandwidth efficiency, these sections of the screen which are updated, referred to as *tiles*, are transmitted in compressed form. 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. Also, if the geometric size of the updated region of the screen is known *a priori*, the amount of network traffic generated is likely to suggest what the screen update was as the image compression algorithm is likely to assign unique data sizes to different image tiles of the same geometric size. In this evaluation, the ability for an adversary to determine which character was typed into a text editor over SSH tunneled VNC is evaluated. Specifically, the dataset consisted of six iterations of all twenty-six lowercase letters typed into the text editor with the data split into 50% *training* and 50% *testing*.

1) Data Gathering Layer: The setup for this network security experiment is as follows. A Docker container representing the *remote desktop* runs the *Xvnc* server as well as the *openssh* server. This container also runs the *Geany* text editor so that a connecting client may be able to 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

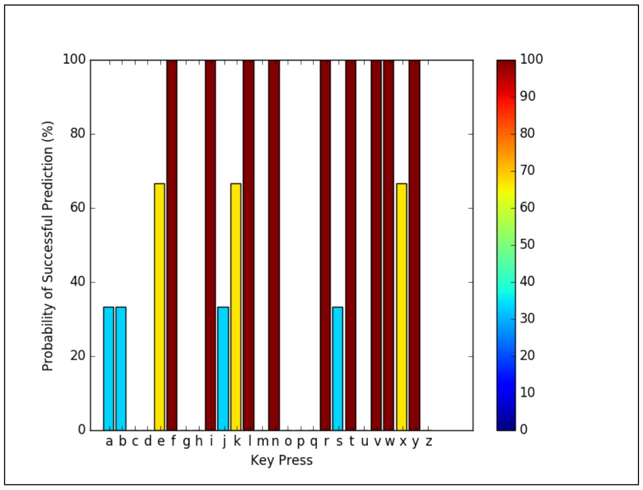


Fig. 5. The probability of success of a wire-tapping adversary predicting keys typed based on observing encrypted network traffic features.

is received by the *openssh* client where it is sent over SSH tunnel to the Docker container running *Xvnc*. 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.

2) Feature Extraction Layer: In order to build the required dataset mapping *GUI interaction* to generated encrypted network traffic, the *feature extraction layer* is employed to fill a dictionary mapping *characters typed* to byte sizes of generated encrypted network traffic.

3) 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.

4) Threat Modeling Layer: 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.

5) Reactive Layer: The *reactive layer* for this scenario simply draws a bar graph describing the probability of successful keypress prediction by a wire-tapping adversary (Fig. 5).

The results of this evaluation have shown that for certain applications, relying exclusively on SSH tunnels for protecting VNC connections can be highly insufficient. This example provides, therefore, a starting point for designing and implementing test scenarios for verifying the security properties of using a *graphical* application over SSH protected VNC.

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 build datasets of interactions with applications and the side-channel cues that provide insight into these interactions. These datasets would include both web application traffic, as well as network activity from IoT devices. 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-2013miroslavstamparsqlmapunderthehood.

[2] D. Genkin, M. Pattani, R. Schuster, and E. Tromer, "Synesthesia: Detecting screen content via remote acoustic side channels," *CoRR*, vol. abs/1809.02629, 2018. [Online]. Available: http://arxiv.org/abs/1809.02629.

[3] H. Wang, M. Brisfors, S. Forsmark, and E. Dubrova, "How diversity affects deep-learning side-channel attacks," *IACR Cryptology ePrint Archive*, vol. 2019, p. 664, 2019.

[4] M. Lescisin, "Sidechannel-detection-framework," Jun. 2019. [Online]. Available: https://github.com/IntegralProgrammer/Sidechannel-Detection-Framework.

[5] R. Spreitzer, V. Moonsamy, T. Korak, and S. Mangard, "Systematic classification of side-channel attacks: A case study for mobile devices," *IEEE Communications Surveys Tutorials*, vol. 20, no. 1, pp. 465–488, 2017.

[6] N. Chawla, A. Singh, M. Kar, and S. Mukhopadhyay, "Application inference using machine learning based side channel analysis," *CoRR*, vol. abs/1907.04428, 2019. [Online]. Available: http://arxiv.org/abs/1907.04428.

[7] M. Li, Y. Meng, J. Liu, H. Zhu, X. Liang, Y. Liu, and N. Ruan, "When csi meets public wifi: Inferring your mobile phone password via wifi signals," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: ACM, 2016, pp. 1068–1079. [Online]. Available: http://doi.acm.org/10.1145/2976749.2978397.

[8] V. Duddu, D. Samanta, D. V. Rao, and V. E. Balas, "Stealing neural networks via timing side channels," *CoRR*, vol. abs/1812.11720, 2018. [Online]. Available: http://arxiv.org/abs/1812.11720.

[9] A. Tannous, J. Trostle, M. Hassan, S. E. McLaughlin, and T. Jaeger, "New side channels targeted at passwords," in *2008 Annual Computer Security Applications Conference (ACSAC)*, Dec 2008, pp. 45–54.

[10] Y. Yarom and K. Falkner, "Flush+reload: A high resolution, low noise, l3 cache side-channel attack," in *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, 2014, pp. 719–732. [Online]. Available: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom.

[11] 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 *Proceedings of the 17th ACM Conference on Computer and Communications Security*, ser. CCS '10. New York, NY, USA: ACM, 2010, pp. 595–606. [Online]. Available: http://doi.acm.org/10.1145/1866307.1866374.

[12] D. X. Song, D. Wagner, and X. Tian, "Timing analysis of keystrokes and timing attacks on ssh," in *Proceedings of the 10th Conference on USENIX Security Symposium - Volume 10*, ser. SSYM'01. Berkeley, CA, USA: USENIX Association, 2001. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251327.1251352.

[13] M. Lescisin and Q. Mahmoud, "Tools for active and passive network side-channel detection for web applications," in *12th USENIX Workshop on Offensive Technologies (WOOT 18)*. Baltimore, MD: USENIX Association, 2018. [Online]. Available: https://www.usenix.org/conference/woot18/presentation/lescisin.