Leveraging USB to Establish Host Identity Using Commodity Devices

Adam Bates, Ryan Leonard, Hannah Pruse, Daniel Lowd, and Kevin R. B. Butler
Department of Computer and Information Science
University of Oregon, Eugene, OR
{amb, ryan, hpruse, lowd, butler}@cs.uoregon.edu

Abstract—**Determining a computer's identity is a challenge of critical importance to users wishing to ensure that they are interacting with the correct system; it is also extremely valuable to forensics investigators. However, even hosts that contain trusted computing hardware to establish identity can be defeated by relay and impersonation attacks. In this paper, we consider how to leverage the virtually ubiquitous USB interface to uniquely identify computers based on the characteristics of their hardware, firmware, and software stacks. We collect USB data on a corpus of over 250 machines with a variety of hardware and software configurations, and through machine learning classification techniques we demonstrate that, given a period of observation on the order of tenths of a second, we can differentiate hosts based on a variety of attributes such as operating system, manufacturer, and model with upwards of 90% accuracy. Over longer periods of observation on the order of minutes, we demonstrate the ability to distinguish between hosts that are seemingly identical; using Random Forest classification and statistical analysis, we generate fingerprints that can be used to uniquely and consistently identify 70% of a field of 30 machines that share identical OS and hardware specifications. Additionally, we show that we can detect the presence of a hypervisor on a computer with 100% accuracy and that our results are resistant to concept drift, a spoofing attack in which malicious hosts provide fraudulent USB messages, and relaying of commands from other machines. Our techniques are thus generally employable in an easy-to-use and low-cost fashion.**

I. Introduction

Determining the identity of a computer is a necessary precondition for trusting it. However, being able to verify that a machine is actually what it presents itself as is a surprisingly challenging problem. Consider a desktop computer in a corporate office. For that computer's user, it would seem natural to assume that observing physical indicators about the machine, such as seeing it is still there, it is the expected make and model, and even that it has the correct serial number on it, should be sufficient to have confidence that the machine is the user's own and is responding to the user's commands. However, even if the computer is physically present, there is little preventing that machine from silently relaying commands to another computer that is in turn providing responses in place of the machine being used. Such an attack against trusted hardware was proposed by Parno [1], who termed it the "cuckoo attack" after the birds that lay their eggs in the nests of other species.

More generally, evading identification through *relay* or *wormhole* attacks has been demonstrated in wireless networks [2], with RFIDs [3], and even with mobile devices using near-field communication [4]. Solutions to the unique identification problem have run the gamut from displaying visual indicators on machines [5] to establishing uniqueness in processor chipsets through physically unclonable functions (PUFs) [6], to relying on trusted hardware such as the Trusted Platform Module (TPM) [7] containing a unique private key. However, as Parno showed, even trusted hardware has the potential to be subverted. Additionally, solutions such as PUFs and TPMs do not address the vast set of legacy systems in place that do not have these devices deployed.

In this paper, we propose that the USB port found on virtually every computer and many embedded devices can be used as a means of determining identity. The combination of software variation among USB stacks, differences in USB host controllers, and variations in manufacture of the bus, chipsets, and other hardware portions allow us to be able to *fingerprint* machines based on the timing of USB messages they send to a device connecting to them. Consequently, we call our technique *USB Fingerprinting*, and in this work we demonstrate its practicality and efficacy for a variety of applications. Using machine learning classification techniques, we demonstrate that, through observing the enumeration phase of the USB protocol, we can differentiate hosts based on attributes such as operating system, manufacturer, and model, and in many cases develop a classification model that uniquely identifies host machines. Our machine identification trials show that USB Fingerprinting, at best, can build a set of models that consistently identifies 70% of a field of 30 seemingly identical machines. However, even these results are strong enough to serve as a reference point in a larger forensic scheme.

Previously, USB forensics has required manual collection of data via an expensive USB analyzer [8], or a the design of a custom embedded device [9]. In contrast, we have developed an automated data collection and protocol analyzer application for a commodity Android smartphone. We confirm that our platform provides equivalent results to USB analyzers, and additionally that our techniques are applicable to even low-cost embedded devices such as Gumstix [10].

NDSS '14, 23-26 February 2014, San Diego, CA, USA
Copyright 2014 Internet Society, ISBN 1-891562-35-5
http://dx.doi.org/doi-info-to-be-provided-later

The ability to perform USB fingerprinting was postulated in a previous workshop publication [8], but this work suffered from a small data corpus and classification techniques that failed to establish even basic information about a host's identity, such as machine model. Other works that have considered USB information focused on the protocol as a vector for attack [9], [11]. Crucially, these previous works, and more generally, most other fingerprinting proposals (e.g., [12], [13], [14]) have made the assumption of a benign environment in which the fingerprint target's input is essentially trusted and no countermeasures are taken in case the host is responding maliciously. For example, Eckersly's proposal for website identification [13] tests against existing browser anonymization plug-ins, but the fingerprint scheme is still ultimately built on trusted input from the target. By contrast, we demonstrate that our approach is robust against a number of active attacks, such as malicious inputs, making it potentially viable as a fingerprinting mechanism in malicious environments. We also demonstrate resistance to concept drift[1] in our approach by examining data measured three months after the original measurements were taken.

USB Fingerprinting identifies devices via a physical connection, rather than relying on wireless [15] or visual indicators [16]. While remote attestation is often desirable, reliance on networked environments is also a limitation. For example, network fingerprinting is not an option for non-networked embedded devices or "sneaker nets", which have recently gained notoriety for having been targeted by the Stuxnet virus [17]. Physical connectivity also prevents spoofing and anonymization attacks to which networking fingerprinting techniques are vulnerable [18], [14], [19].

This paper makes the following contributions:

- **A methodology for feature extraction of USB trace data:** We examine USB *enumerations*, handshakes representing milliseconds of communication between master and slave devices. Applying this insight, we develop a feature extraction methodology that we apply to tens of thousands of traces. We conduct a university-wide survey of USB stack behavior, presenting a dataset of over 30,000 traces of USB operations from 256 different machines of various makes, models, and operating systems. We quantify the information gain of the resultant feature vector across a variety of class labels representing different host machine attributes.
- **Design and evaluation of USB-based machine classification techniques:** We use USB data to differentiate between different manufacturer models with 97% accuracy, between operating systems with 100% accuracy, and between OS versions with an accuracy of 94%. Even more remarkably, we employ statistical techniques that allow us to iteratively refine results for individual machines, yielding a fingerprint that can uniquely identify 70% of a field of 30 identically specified machines. We analyze our feature vectors and machine learning models in order to discover methods of further optimization. We also show that our techniques apply to USB-equipped embedded devices, and can be used to swiftly detect the presence of virtualization. We consider the efficacy of our approach against an active adversary, and discover that our scheme is resistant to IP-based relay attacks, and spoofing attacks that would prove successful against previous USB fingerprinting techniques [8], [11], [9].
- **Development of collection and analysis tools for commodity deployment:** While USB analyzers can be used to get fine-grained protocol information, they suffer from being bulky, expensive, and uncommon. To make USB Fingerprinting broadly adoptable, we develop and release smartphone applications for the Android operating system that can be used to effectively and automatically perform the collection and analysis of USB data. Both the applications and dataset are to be released, with continual updates as our corpus grows. To our knowledge, this is the largest compilation of such data ever to have been made publically available.

[1]Concept Drift [20] is a machine learning problem in which statistical properties (e.g., accuracy) of a classification model degrade over time.

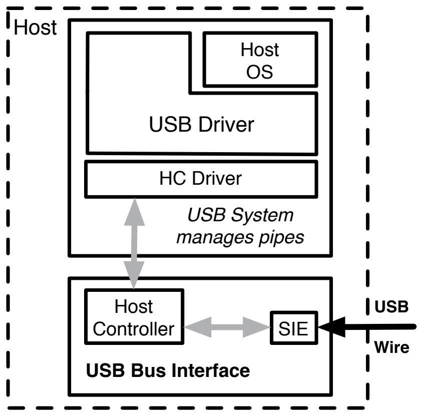


Fig. 1: Overview of a host USB Stack.

The rest of the paper is structured as follows: Section 2 provides an overview of USB operation and the specific messages that we collect, Section 3 describes our methodology for collecting USB trace data, and Section 4 describes our classification techniques and results. Section 5 describes the applicability of USB Fingerprinting to other contexts, such as detecting virtualized environments, fingerprinting devices that are not computers, using multiple collection devices, and issues relating to concept drift. Section 7 describes how USB Fingerprinting can be widely deployed with our developed Android application, explores how we can defend against fraudulent and relaying hosts, and discusses future directions. Section 8 provides related work and Section 9 concludes.

II. USB Protocol

A. Overview of Operation

USB is a standard that defines a software protocol, firmware protocol, and a set of hardware used in communication between a *host* and a *device* across a *serial bus* [21]. Since USB is a master/slave protocol, the host initiates all interactions. As shown in Figure 1, USB stacks vary from

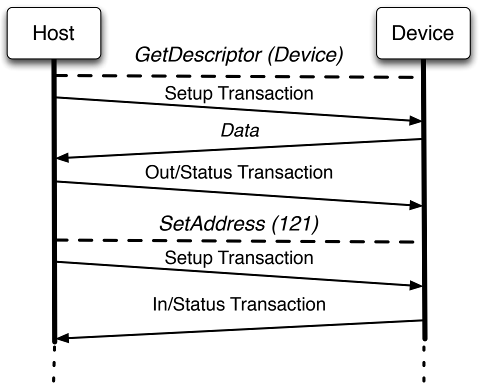


Fig. 2: An example USB data flow. Control transfers (dotted lines) are comprised by a set of transactions (solid lines).

host to host, and are made up of a host controller, a controller driver, a USB driver, and host software. Many machines today support USB, including personal computers, servers, tablets, routers, and embedded systems. We focus on the USB 2.0 protocol, since while the USB 3.0 "SuperSpeed" protocol has been codified, it has not been as widely deployed.

In order for a USB device to be used with a given host, it must go through a setup procedure consisting of three steps. First, is the *bus setup*, during which a set of standard electrical signals is relayed between the host and device's respective serial interface engines (SIE). This step indicates to the host that a device is connected; the two parties then handshake and negotiate parameters such as the communication speed of the device. The second step is the *enumeration* phase, whereby the host queries the device to determine information such as the device's type (e.g., mass storage, human interface device), manufacturer and model, and the functionality it supports, among other parameters. Finally, further interactions are passed from the host's client software through the standard system call interface (e.g., *read(), write(), ioctl()*) to the device's high-level USB functions (e.g., providing an interface to internal storage, relaying video from a webcam).

B. Enumeration

In this work, we analyze the *enumeration* phase of a USB interaction to make inferences about a host's USB stack. Enumeration is a good candidate for analysis, as extensive interaction with the host system is not required to force the host to enumerate a device. Thus, it is possible to trigger USB enumerations on any physically accessible machine, even when lacking login credentials. Additionally, the enumeration phase is well defined in the USB 2.0 specification [21], making it easy to interpret. While specific message content and timing will vary depending on the host's USB stack and the connecting peripheral, the presence and purpose of the enumeration phase is host and device agnostic.

The process of enumeration is a host-driven operation that consists of a three-layered protocol. At the top layer are *control transfers*, USB data flows that offer lossless delivery, that exchange configuration information between the host and the device endpoint. An example of a control transfer is `GetDescriptor(String Manufacturer)`, which informs the host of the device's manufacturer. At the middle layer, *transactions* offer a logical abstraction for bundles of packets. A notable transaction is the **setup transaction**, which describes in detail what the following transactions will be; once the content of the setup transaction is known, the content of subsequent messages is well defined. Note that interrupt requests (IRQ) on a device must be signaled at the end of each transaction, to inform the device's software to queue a future transaction. At the bottom layer there are USB *packets*, which transmit the actual data. Each control transfer is formed by two or more transactions, and each transaction is composed of two or more packets. A sample composition of USB enumeration is portrayed in Figure 2: the control transfers `GetDescriptor` and `SetAddress` appear in the figure, and are composed of transactions that include *setup* and *out/status*.

III. Methodology

This work is inspired by the advent of non-intrusive, high-speed USB protocol analyzers. These tools passively intercept individual packets in order to reconstruct USB traces at the logical layer. Developers are able to use this information to detect errors at different levels of the USB stack as well as measure performance [22]. Unfortunately, these devices are costly, reaching into the thousands of dollars. In this study, we worked with a USB analyzer that weighed 1.65 pounds with 6x5x2.5 inch dimensions, hampering its portability and adoptability. In order to inspect the results of the USB trace, users must also plug the USB analyzer into another PC with specialized software. Additionally, it is difficult to collect large numbers of USB traces, as manual unplugging and reinsertion of the device is required.

This section is structured as follows: in Section III-A we present a threat model for our scheme; in Section III-B we compare the performance of an industry-class USB protocol analyzer with our own Android-based device; in Section III-C we detail our data collection procedure; in Section III-D we discuss the contents of our corpus of collected data; in Section III-E we describe our feature extraction procedures; and in Section III-F we offer insights into how these features can be used to identify hosts.

A. Threat Model

We assume that an adversary has the ability to arbitrarily control the software of the host system being fingerprinted, and has ownership of the kernel. We assume that the adversary thus has the ability to modify the USB device driver and to modify USB messages returned from the host. Additionally, such ownership allows the host to arbitrarily *relay* messages to a different computer through a network interface, thus allowing the bypass of the LPC bus where the TPM resides, and other peripherals. We assume that the attacker has the ability to physically access the host machine and to tamper with it; however, we assume that the initial collection of measurements is performed on an uncompromised machine. We do not assume the use of wireless USB on the system. We further analyze the security of the scheme and its potential limitations in Section VI-B.

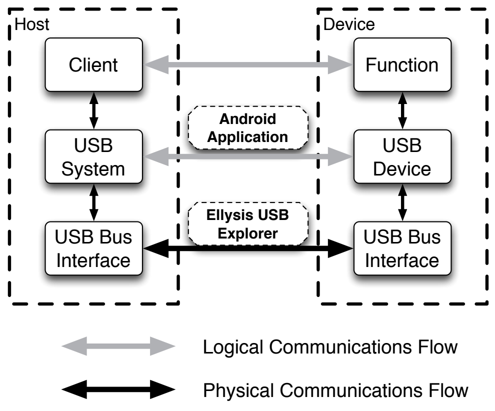


Fig. 3: USB communications model. We observed physical communications via an *Ellisys USB Explorer* to confirm the correctness of an Android-based protocol analyzer.

Control Transfer	Android (sec)	Ellisys (sec)
GetDescriptor (Device)	0	0
SetAddress (121)	N/R	0.111 992 200
GetDescriptor (Device)	0.132 202	0.131 983 683
GetDescriptor (Configuration)	0.132 508	0.132 773 683
GetDescriptor (Configuration)	0.132 782	0.132 890 216
GetDescriptor (String: Language)	0.133 057	0.133 015 483
GetDescriptor (String: Product)	0.133 423	0.133 257 716
GetDescriptor (String: Manufacturer)	0.133 698	0.133 390 216
GetDescriptor (String: Serial Number)	0.134 003	0.133 508 250
SetConfiguration	0.134 491	0.133 803 183
GetDescriptor (String: Interface)	0.134 918	0.134 272 716
ClearFeature (Endpoint 1 IN)	N/R	0.165 663 866
ClearFeature (Endpoint 2 OUT)	N/R	0.166 395 233
ClearFeature (Endpoint 1 IN)	N/R	0.167 016 066
GetDescriptor (String: Language)	1.135 742	1.134 755 000
GetDescriptor (String: Serial Number)	1.136 048	1.135 822 016

TABLE I: Enumeration on a Linux host. The Ellysis USB Explorer confirms the approximated timing information captured by our application in the Android kernel. N/R denotes events that were not recorded by our device.

B. Hardware USB Analyzer

Our approach to machine fingerprinting begins with the observation that lightweight devices with USB interfaces are now ubiquitous, to the point that many of us carry multiple USB devices on our person throughout the day. Smart phones charge and physically connect to other machines with USB, and USB flash drives are a fixture of key chains and briefcases in many work environments. Even certain USB drives, such as Imation's IronKey [23], now come equipped with on-board CPUs that provide additional security and usability features.

As we developed our own protocol analyzer device, we used the Ellisys USB Explorer 200 [22] to establish a ground truth for USB observation. Figure 3 visualizes the Ellisys explorer's ideal vantage point in the USB communications model; by physically intercepting packets as they traverse the wire, Ellisys can obtain highly precise timing information and perfectly reconstruct traces. This performance comes at the cost of usability. Figure 3 also depicts the vantage point of our own USB collection mechanism, an Android smartphone application that intercepts messages at the device controller driver level. By routing our own device through the USB analyzer, we were able to confirm the correct behavior of our Android application. Using Ellisys software, we were able to map the timing data of our observed USB enumeration recordings to the actual serial bus activity. A representative mapping between Android and Ellisys data is shown in Table I. This assured the proper functionality of our USB collection application, allowing us to move forward with data collection.

C. Smartphone Collection

To avoid using costly and exotic hardware, we developed a USB analyzer for a device that many users already own, namely an Android smartphone. The timing data measurable from the device's userspace USB function proved too coarse-grained, so we resorted to accessing kernel memory through rooting the phone. We used kprobe modules to dynamically alter *musb_gadget_ep0.c (forward_ to_driver*, *musb_g_ep0_irq)* and *android.c (android_setup)*. With these modifications installed, we were able to capture individual IRQs with microsecond timing. We were also able to force calls to *usb_disconnect* after each enumeration, thereby automating the process of recording multiple traces on a target host. This allowed us to collect the individual transactions that constitute the full enumeration process. While the need to root the device is a limitation to our approach, we note that the installation of aftermarket open-source firmware for Android devices is now a simple process thanks to communities like CyanogenMod [24]. Moreover, when finished collecting data, the kprobe modules are automatically uninstalled, allowing the phone to resume the standard USB behaviors of a stock Android device.

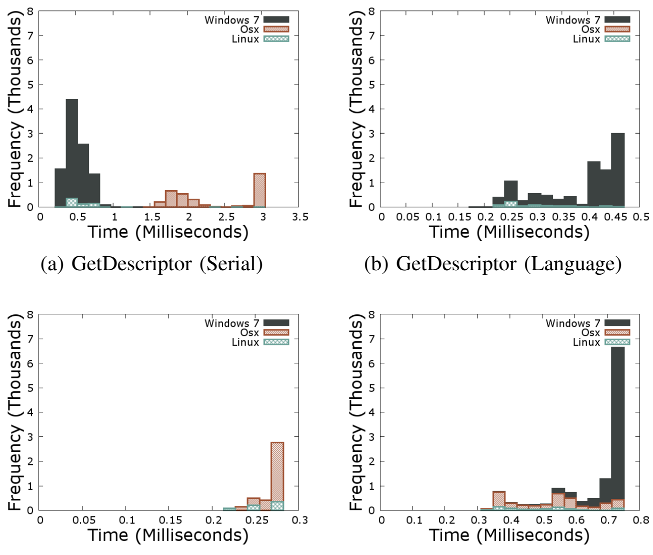
To ensure consistent behavior across a variety of hosts and devices, we chose to collect timestamps from all transactions' IRQs, but the data payloads of only the setup IRQs. We found that logging the contents of every IRQ required excessive memory copying in atomic functions, consequently causing USB enumeration to fail and the device's kernel to panic. Additionally, the setup transaction dictates the contents of subsequent transactions within the control transfer. We concluded that dumping the payloads of setup transactions alone would offer sufficient information to see the full context of enumeration. We call the combination of transaction timestamps and content captured over the course of enumeration a *USB Trace*.

Kernel-level buffering occurred whenever an IRQ's timestamp was logged. The resulting average delay was 120 μs compared to the serial bus activity observed via Ellisys. For example, the delay between the first and second IRQ in a control transfer is as short as a 3 μs. However, the log showed as much as 150 μs delay, a 147 μs discrepancy. Fortunately, at the end of each control transfer, there is a pause of sufficient length to permit the buffer time to fully clear, thus resynchronizing with the serial bus. This gives us accurate timing information for the setup transaction of every control transfer, shown in Table I.

Using the *Android USB Analyzer* application that we explain in greater detail in Section VI-A, we were able to automate collection of USB traces. The process for data collection was as follows:

Class	Label	Host Count
OS		256
	Linux 3.2	8
	OSX 10.6	2
	OSX 10.7	15
	OSX 10.8	49
	Windows 7 SP1	182
Model		256
	Apple iMac 10	27
	Apple iMac 11	5
	Apple iMac 12	25
	Apple iMac 13	8
	Apple Mac Mini 52	2
	Dell Dimension 4700	2
	Dell Latitude 6500	23
	Dell Latitude 6510	3
	Dell Optiplex 745	32
	Dell Optiplex 760	6
	Dell Optiplex 980	34
	Dell Optiplex 990	43
	Dell Optiplex gx520	4
	Dell Optiplex sx280	5
	Dell Precision t3500	32
	Dell Precision t3600	5
TOTAL MACHINES INSPECTED:		256
TOTAL MEASUREMENTS COLLECTED:		32,150

TABLE II: Description of data corpus.



(a) GetDescriptor (Serial) (b) GetDescriptor (Language)

(c) GetDescriptor (Manufacturer) (d) SetConfiguration (1)

Fig. 4: Timing data for selected features by operating system.

1. Record target machine attributes in Android USB Analyzer interface (e.g., serial number, OS, Manufacturer).
2. Hard reset the target machine.
3. Disconnect other USB devices from the target machine.
4. Plug the phone into the target machine.
5. Allow the Android USB Analyzer application to automatically collect the specified number of USB enumerations.

The procedure was meant to eliminate additional variables that might influence the timing result. Determining the robustness of the approach when the machine is potentially under load, or identifying appropriate periods of machine quiescence, is future work.

We elected to develop on Android due to the widespread adoption of this platform. However, we note that our methodology would be similarly effective using other collection devices. For example, in preliminary tests we collected data using a Gumstix device [10] running a Linux kernel that was modified to record USB message timestamps. Using the Gumstix data, we identified machines by OS and model with comparable accuracy to our Android data corpus (see Section IV-B).

D. Data Corpus

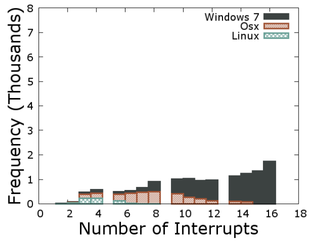
Following the procedure described above, we performed data collection on a variety of machines across a university campus. In addition to 8 student-accessible computer labs, we also obtained decommissioned machines from the university's IT department. Our dataset includes thousands of traces collected from 6 Linux, 66 OSX, and 182 Windows hosts. The corpus is described comprehensively in Table II. We collected at least 50 enumeration traces from each machine, and collected data from some machines multiple times, resulting in over 30,000 traces collected. We continue to add to this dataset, and expect it to eventually describe the USB characteristics of tens of thousands of machines.

During data collection for this study, we were limited by the homogeneity of the computer labs on our campus. For example, all Windows 7 systems in university-administrated labs were based on the same disk image. The use of decommissioned machines allowed us to supplement the university-controlled hosts. For each machine class label, we collected data from at least 2 machines, such that one machine could be withheld for evaluation purposes.

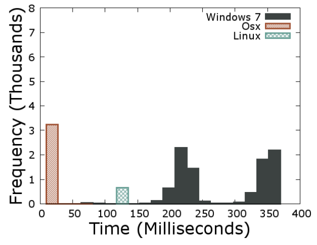
E. Feature Extraction

As a necessary first step to analyzing our data, we performed a variety of pre-processing tasks that extracted meaningful information from each trace. The combined output of these efforts was a per-trace feature vector that we used as inputs to machine learning classification algorithms. We constructed features that, based on our knowledge of the USB enumeration phase, were likely to be stable and effective at discriminating between different classes of machines. Initially, each enumeration trace was a series of interrupts corresponding to `setup`, `out/status`, `in/status`, and `idle` IRQs. Our first set of features captured control transfer timing information. Control transfers were identified using the `setup` IRQ, whose contents specifies the subsequent control transfer. Control transfer responses vary in length, so we further divided each control transfer by the size (in bytes) of the requested response. For each control transfer feature, a time value was assigned that was equal to the duration between the transfer's *setup* IRQ and the *setup* IRQ of the subsequent transfer.

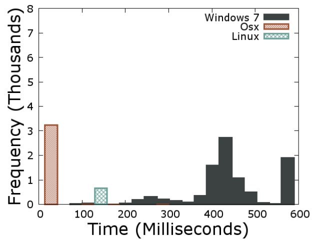
Visual inspection of these features demonstrates the intuition behind our fingerprinting approach. The frequency histograms for a representative subset of control transfer features are depicted in Figures 4a-4d, and are plotted by their



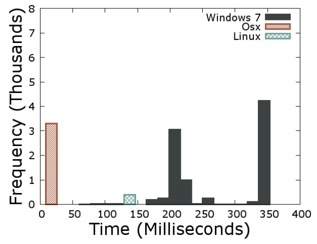
(a) Number of Idle IRQs.



(b) Average Inter-IRQ Gap.



(c) Length of Enumeration.



(d) ct_ngram(0;6)

Fig. 5: Timing data for additional features by operating system.

operating system class label. As an exercise, it is possible to differentiate between operating systems with 100% accuracy based on visual inspection of these 4 features alone. OSX can be detected in a single step through the absence of the `GetDescriptor (Language)` transfer (Fig. 4b), and Linux and Windows hosts can be differentiated based on the presence of `GetDescriptor (Manufacturer)` (Fig 4c).

Trace-level statistics were also included in the feature vector. A representative subset of these features is depicted in Figures 5a - 5c. Figure 5c indicates that OSX hosts enumerate much faster than Linux and Windows hosts. The per-trace count of Idle IRQs (Fig 5a) have a shallow distribution, but can also be used to differentiate operating system. We discover in Section V-A that a high frequency of `idle` IRQs is also a strong indicator of the presence of the Xen hypervisor.

Lastly, we attempted to include the order of the messages in the trace at both the IRQ and control transfers level. However, we found high variance in the ordering of IRQs between traces, even when comparing traces from the same machine. We also found that a feature describing the ordering of control transfers would offer little discriminating power over our existing features, as the string would primarily denote the absence or presence of certain control transfers. Rather than explicitly describe the sequence of messages, we chose to include a family of features that clustered timing information for sequences of messages. We summed the timing information of every permissible sequence of messages in the trace, naming each for its starting position and length, e.g., *ct-gram(1,4)* represented the duration of 4 consecutive control transfers starting at transfer 1. We repeated this process at the IRQ level, adopting the naming convention *irq-gram(·)*. A representative n-gram feature is depicted in Figure 5d.

Combined, these measurements offered a vector of 152 features. We used a set of Python scripts to parse individual traces, extract the features, and output them into a single CSV file. This file was later converted to Attribute-Relation File Format (ARFF) for use with the Weka software suite.

Feature	**Information Gain**
`Length of Enumeration`	3.4 bits
`GetDescriptor(Interface)`	2.6 bits
`ct_ngram(0;6)`	2.2 bits
`Average Inter-IRQ Gap`	2.2 bits
`ct_ngram(0;3)`	2.1 bits

TABLE III: The 5 highest ranked features by information gain. Class labels were assigned by machine for these measurements.

F. *Feature Inspection*

Prior to classification, we attempted to gain further insight into the discriminating power of our feature vector. To accomplish this, we calculated the *information gain* of each individual feature. Information gain is a measure of the reduction in entropy achieved by learning the output of a given random variable. For a function $F(\cdot)$ with the probability density function $P(f_n), n \in [0, 1, ..., N]$, entropy is given by the formula $H(F) = -\sum_{n=0}^{N} P(f_n)\log_2(P(f_n))$. For a class X and a given feature output $F(X) = f_n$, information gain (reduction in entropy) is calculated as $I(X, f_n) = H(X) - H(X|f_n)$. The information gain for our 5 most discriminating features is displayed in Table III. These values represent each feature's information gain in isolation. These measurements were obtained based on assigning each trace a class label corresponding to the machine from which it was collected. Despite the difficulty of conceptualizing the usefulness of our n-gram features, we see that they can be highly prioritized inputs to machine learning classification algorithms, notably `ct_ngram(0;6)` and `ct_ngram(0;3)`.

IV. Classification

In this section, we undertake a variety of classification challenges to explore the discriminating power of USB enumeration data. We do so by employing multiple machine learning classifiers, described in Section IV-A, to predict different characteristics of the origin machines to which our device enumerated. In Section IV-B, we build on past work in using USB to identify machine attributes such as OS and manufacturer model. In Section IV-C we go a step further, developing a fingerprint that will uniquely identify an individual machine amidst a field of identically specified machines.

A. *Classifier Survey*

In an effort to maximize accuracy and ensure the robustness of our models, we analyzed our data with several different supervised learning classifiers. Supervised learning algorithms generate an inferred function (or *model*) to classify previously unseen data instances. They are built with a set of training data instances that contain a vector of attributes, as well as a class label. Supervised algorithms then analyze this data, outputting a classification model. We used the popular Weka libraries [25], which are well-respected tools in the machine learning community.

The results of the full survey have been omitted for brevity, but can be found in our technical report [26]. Briefly,

OS Version	Accuracy
Linux 3.2	100%
OSX 10.6	100%
OSX 10.7	68%
OSX 10.8	86%
Windows 7	100%

TABLE IV: OS Version accuracies by class label.

Model #	OS Ver.	Host Ctrl	HC Drvr	Count
iMac 12	OSX 10.7	Intel C200	5.1.0	14
iMac 12	OSX 10.8	Intel C200	5.4.0	11
iMac 13	OSX 10.8	Intel C200	5.5.0	8

TABLE V: iMacs that shared a common host controller in our dataset.

we experimented with a variety of classifiers on a preliminary dataset, including Random Forests [27], J48 decision trees [28], Decision Stump Boosting [29], Support Vector Machines (SVMs), and Instance-Based Learners (IBLs). We found that the decision trees classifiers were an excellent fit for our dataset, particularly because individual features were often sufficient to rule out large subsets of the possible class labels. SVMs and IBLs, while effective in some of our tests, were prone to *the curse of dimensionality* and thus only operated effectively with smaller feature vectors. We ultimately chose to make use of the Random Forest classifier, whose boosting method of iterative model building is well known for producing accurate results [30].

Next, we turned our attention to identifying machine models. Using the same approach, the classifier generated a model that achieved 97% accuracy over the test data. Accuracies by class label are contained in Table VI. The classifier was extremely effective at identifying all models except for the Mac Mini, which was undervalued due to representing less than 1% of the dataset.

B. Host Attribute Identification

For our initial investigation, we used single USB enumerations to predict a number of host machine attributes: operating system family, operating system version, machine model, and model number; an example set of attributes is *OSX*, *OSX 10.8*, *Apple iMac*, and *Apple iMac 13*, respectively. Stavrou et al. formulate host attribute identification as the first step in a variety of phone-to-computer attacks, allowing the adversary's phone to exploit specific system vulnerabilities while avoiding the need for brute force approaches [11]. Each of these attributes were formulated as a separate machine learning problem, and for each problem we used the Random Forest classifier. To test the parameterization of the classifier, we used a separate preliminary dataset of a small collection of machines. We selected a forest size of 20, leaving other parameters at the default Weka settings. We found that these parameters were robust across all of our host attribute identification experiments. Following the default Weka parameters, the model was built on 66% of the dataset, with 34% withheld for evaluation. The data was partitioned such that traces from a specific machine would appear in either the test set or training set, but not both. Percentage splits are an alternative evaluation method to cross-validation,

Model	Accuracy
Apple iMac	100%
Apple Mac Mini	88%
Dell Dimension	96%
Dell Latitude	94%
Dell Optiplex	99%
Dell Precision	95%

TABLE VI: Machine Model accuracies by class label.

MNF	Model	Number	Accuracy
Apple	iMac	10	100%
Apple	iMac	11	76%
Apple	iMac	12	92%
Apple	iMac	13	65%
Apple	Mac Mini	52	90%
Dell	Dimension	4700	34%
Dell	Latitude	e6500	91%
Dell	Latitude	e6510	96%
Dell	Optiplex	745	99%
Dell	Optiplex	760	95%
Dell	Optiplex	990	95%
Dell	Optiplex	980	96%
Dell	Optiplex	gx520	41%
Dell	Optiplex	sx520	57%
Dell	Precision	t3500	95%
Dell	Precision	t3600	64%

TABLE VII: Model Number accuracies by class label.

and are actually more conservative than the popular 10-fold cross validation strategy in that they provide less training data to the classifier.

As indicated previously, detection of operating system family proved to be a trivial task using our feature vector, and in fact the model was able to predict the OS of the test dataset traces with 100% accuracy. In addition to the features that we used in our example from Section III-E, the classifier determined that the timing data of `ct_ngram(0;2)` and a variety of `irq_ngram`s offered high information gain. What these features had in common is that they all indicated that OSX traces were the fastest, and that Linux traces were consistently faster than Windows traces.

Using the same approach, we then built a model for operating system by version. When evaluated against the test data, this model achieved 94% accuracy. The cause for this loss of accuracy is apparent in Table IV. The model struggled in differentiating OSX 10.7 from OSX 10.8. This struggle is explained through reviewing the iMac USB stack information, contained in Table V. Apple iMacs 12 and 13 share a common host controller, and the classifier prioritized OSX 10.8, which appears 3 times more frequently in our training set. We note that reweighting our training set would improve the accuracy balance between these predictions.

Finally, we identified the specific model number of a host machine. In part due to redundant USB stack components, previous work has failed to offer a method of differentiating hosts at this level of granularity [8]. In contrast, our model achieves 90% accuracy. Accuracies by label are pictured in Table VII. Predictably, accuracy suffered for classes that were underrepresented in the dataset, such as the Dell Dimension 4700 and Optiplex sx520 (See Table II). Notably, there were no crossover predictions between the Dell and Apple machines.

Host	Acc	Trace Req	Sys Time	Real Time
`iMac 1-2`	56%	150 traces	38 sec	5 min
`iMac 3`	64%	250 traces	63 sec	8 min
`iMac 4`	56%	450 traces	113 sec	15 min
`iMac 5-9`	55-58%	N/A	N/A	N/A
`PC 1-13`	60-86%	150 traces	38 sec	5 min
`PC 14-21`	68-80%	250 traces	63 sec	8 min
`PC 22-30`	53-72%	N/A	N/A	N/A

TABLE VIII: Results for machine identification. **Acc** is the accuracy of individual trace predictions. **Trace Req** shows the number of traces required to reach 95% confidence of the host's identity using a X^2 test. **Sys Time** corresponds to the total amount of USB activity observed, which is the theoretical lower bound on the time required to achieve these results. **Real Time** describes the time required to collect the traces on our unoptimized device.

C. *Machine Identification*

We next set out to see if our feature vector was sufficiently expressive to differentiate between identically specified machines. For this trial, we inspected 2 datasets from different campus labs: one that contained 9 brand new Apple iMac 13.2s, and another that contained 30 Dell Optiplex 745s. Due to the increased difficulty of this problem, we re-labeled our dataset with binary classes such that the Random Forest classifier generated a model to answer the following question: *Does this trace belong to the* **target** *machine, or some* **other** *machine?* Accordingly, we created a total of 39 different models, one for each `iMac` or `PC`. In previous experiments using a separate preliminary dataset, we determined that a forest size of 125 and a maximum tree depth of 3 was an effective configuration for this problem. The other classification parameters remained at their default settings in Weka. Once again, our datasets were split 66%/34% for the respective training and test data. In order to prevent the Random Forest boosting mechanism from over-prioritizing the more frequently-appearing **other** label, the datasets were reweighted such that both labels were equally prevalent. These accuracies are pictured in the **Acc** column of Table VIII; our ability to identify the origin machine of a trace ranged from 53% to 80% in accuracy depending on machine.

Due to noise and variability, multiple traces from the same machine were often classified differently. We hypothesized that, by combining the classification results of multiple samples, we would be able to reduce the error rate of our prediction. We would like to choose the label that is predicted most often as the sample count approaches infinity. If that label correctly identifies the machine from which the traces originated, we know that we have overcome the variance in our data, and that our model is truly able to identify the machine. If this label is incorrect, there is systematic error due to bias, and our model is broken. In practice, the model is also broken if the difference in prediction frequency between the correct and incorrect label is not statistically significant; that is, we have not overcome the sample variance, and classifier's prediction is not much better than a fair coin toss.

First, we determined how many predictions were necessary to be confident that the selected label was significantly more accurate than a fair coin toss. To do so, we applied the Chi-Squared (X^2) statistical test of independence to each model for increasing numbers of traces $T = \{150, 250, 350, 450\}$. We adopted the null hypothesis that, after t traces, the **target** and **other** labels were equally likely to be predicted for a given trace, giving us the expected result $E = \{0.5, 0.5\}$. We then polled the model for t predictions using traces that were drawn from the **target** machine, forming a set of observed results O. The X^2 test statistic is given by the formula $X^2 = \sum_{i=1}^{t} \frac{(O_i - E_i)^2}{E_i}$ where O_i are observed frequencies and E_i are expected frequencies. This statistic can be compared to a critical value from the X^2 distribution that is parameterized to a given confidence level; if the statistic is greater than the critical value, we rejected the null hypothesis that the observed and expected frequencies were drawn from the same underlying distribution. To account for false positives, this process was then repeated using traces from **other**. For each machine, a test that used t traces succeeded if the model predicted the correct label for both cases in each of 20 trials with randomized test inputs.

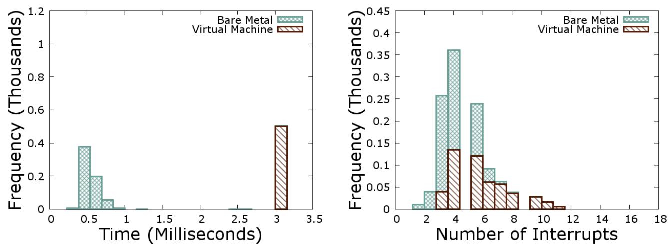
The results of the X^2 test are summarized in the **Trace Req, Sys Time, and Real Time** columns of Table VIII. If the test did not reach 95% confidence with $t = 450$ traces, it was considered a failure and N/A is listed in the row. Otherwise, the minimum number of required traces required to reach 95% confidence is listed. We note that many statistical hypothesis tests could be used in place of X^2 to determine how many predictions are necessary to sufficiently reduce the error rate of the classifier; we also modeled the label predictions as a Bernoulli process, with each trace being an independent Bernoulli trial. Assigning the **target** and **outlier** labels the values 0 and 1 respectively, t trials were performed. If, at the end of the process, the standard error of the sample mean was more than 2 standard deviations (i.e. 95% confidence) away from 0.5, we knew the result to be statistically significant. For each machine, we found that a comparable number of trials was required to the result found using the X^2 test.

These results can be interpreted as follows. At $t = 250$, we predicted machine identities in the Windows corpus with 100% effective accuracy, but just 70% coverage. That is, PCs 1-21 are identified 100% of the time after 250 traces, while PCs 22-30 remain unidentifiable. During the intermittent phases of the test, e.g., at $t = 50$, the accuracy for PCs 1-21 has increased compared to t=1, but are not yet statistically significant according to the X^2 test[2]. Coverage is worse for the iMac corpus, with just 44% coverage with $t = 450$. While these results are encouraging, we conclude that USB Fingerprinting in its present state does not have sufficient coverage for commodity deployment. We discuss how we intend to build on these results in Section VI-C.

V. Analysis

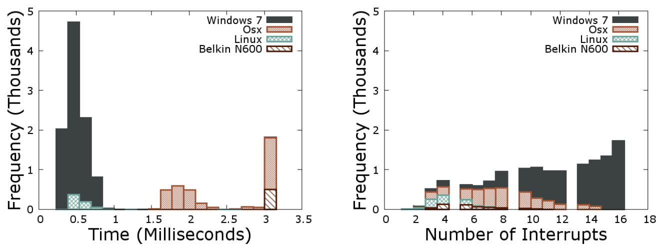
We now perform additional testing to explore a variety of special circumstances and challenges for USB Fingerprinting. First, we demonstrate the ability to detect virtual environments, establishing USB Fingerprinting as a viable detection method for virtual machine-based rootkits. We also show that, in contrast to other fingerprinting methods, USB Fingerprinting is

[2]Note that the values in the **Acc** column of Table VIII represent the expected accuracy of a random trial in which $t = 1$.



(a) GetDescriptor (Serial Number) (b) Number of Idle IRQs

Fig. 6: Control transfer timing data exposes the presence of Virtual Machine Monitors.



(a) GetDescriptor (Serial Number) (b) Number of Idle IRQs

Fig. 7: Timing data for Belkin N600 Router is distinct when compared to other hosts.

a more universal technique that can be applied to a variety of USB-equipped embedded devices. Finally, we challenge the robustness of our USB Fingerprinting classification models against multiple collection devices and concept drift over time.

A. *Virtualization*

One particularly promising application for USB Fingerprinting is as a detection method for virtual machine-based rootkits (VMBRs). VMBRs are difficult to discover with many existing security utilities because they control the host state observed by software, and make almost no alterations to the host state [31]. This makes VMBR detection difficult without a method of physical attestation.

To discover potential USB-based indicators of virtualized environments, we collected trace data from a Xen-enabled Linux system. When the Xen kernel boots, it specifies another kernel in storage to hoist into a virtual machine and run as an administrative domain (dom0) [32]. This allowed us to observe USB enumerations for an unmodified target host running both on bare metal and on top of a virtual machine monitor. The hoisting procedure is actually quite similar to in-the-wild VMBRs that load a hypervisor on-the-fly [33], [34], [31], particularly an exploit that hijacks the Xen hypervisor by executing malicious code in dom0 [35]. Xen exposes a USB host controller to dom0 and does now otherwise interfere with its USB activity. This is in contrast to domUs, which must access USB through passthrough or emulation mechanisms.

Data collection occurred on a Dell PowerEdge R610 server running Xen 4.1. The target host was running the Linux 2.6.40 kernel. We collected 2000 total USB enumerations on the target host, 1000 from bare metal and 1000 from the hardware-virtualized VM. A few of the features are visualized in Figure 6. As evident through visual inspection of these graphs, there are several strong USB indicators of virtualized environments. In fact, when trained on traces from both the bare metal and dom0 hosts, the Random Forest classifier could distinguish between them with 100% accuracy over separate testing data.

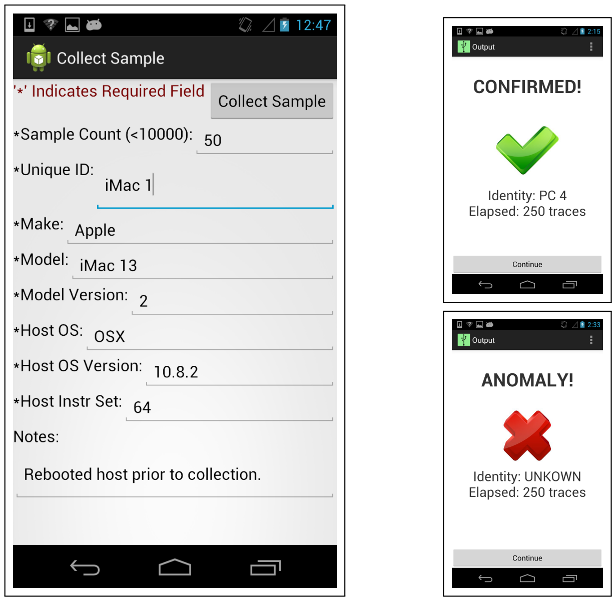
In the event of an actual attack, however, we would not have the luxury of foreknowledge of the VMBR's characteristics. As such, we re-formulated hypervisor detection into an anomaly detection problem. We employed a One Class SVM classifier that was only permitted to train on traces from the bare metal machine. One Class classifiers can fall victim to *the curse of dimensionality*, so in this trial we applied our knowledge from the previous trial to select a compact feature vector. The following features had the highest information gain: `GetDescriptor(SerialNumber)`, `Idle IRQ Count`, `Enumeration Duration`, `Average IRQ Gap`, and `ct_ngram(1,5)`. The SVM used a polynomial kernel, and its nu-value was set such that at least 80% of training traces would fall into the support region. Against the test data, the SVM misclassified just 2.9% of the dom0 traces as falling within the support region. This results in a False Positive rate of 20% and a False Negative rate of 2.9%, making USB Fingerprinting a promising defense against VMBR attacks. As in Section IV-C, the effectiveness of this model can be boosted by taking an ensemble vote of many trace classifications and performing a X^2 test on the result; in 100 out of 100 trials, we successfully distinguished virtualized environments from bare metal using just 13 seconds of observation (50 traces). This finding is consistent with the recent observation that obscuring timing evidence of a hypervisor-based rootkit is extremely difficult [33].

B. *Embedded Device Fingerprinting*

In contrast with other fingerprinting techniques, USB Fingerprinting can be applied to any USB-enabled device that can run in host mode. To demonstrate this, we collected traces from a Belkin N600 wireless router. The router had a USB port intended for use as a plug-and-play media server. Following the same data collection procedure, we collected 500 traces from the router, processed them, and compared them to the features of the rest of our dataset. The router exhibits visibly different timing for many features, some of which are pictured in Figure 7. We re-built the machine model classifier from Section IV-B to include the Belkin N600, and found that the router was correctly identified with 100% precision.

C. *Cross-device Collection*

We are the first study to demonstrate the ability of using USB Fingerprinting to detect minute differences in hardware and software that are present in seemingly identical machines. This being the case, it is reasonable to speculate as to the robustness of our classifiers when data is being collected across multiple slave devices. The act of data collection could possibly be too sensitive for the intermingling of multiple collection devices. Should this be the case, system administrators using


Collect Sample

'*' Indicates Required Field

*Sample Count (<10000): 50

*Unique ID: iMac 1

*Make: Apple

*Model: iMac 13

*Model Version: 2

*Host OS: OSX

*Host OS Version: 10.8.2

*Host Instr Set: 64

Notes: Rebooted host prior to collection.

CONFIRMED!

Identity: PC 4
Elapsed: 250 traces

ANOMALY!

Identity: UNKOWN
Elapsed: 250 traces

Fig. 8: Screenshots from our Android machine fingerprinting apps: attribute entry (left), and machine identity notices (right).

USB Fingerprinting would need to preserve a single collection device for the lifetime of each machine. When this device fails or is lost, they would need to re-fingerprint all machines with a new device. Such a constraint would seriously undermine the adoptability of our approach.

To test classifier robustness across different devices, we collected data from the same machines using two identical Samsung Galaxy Nexus phones. The phones both collected traces for each machine within minutes of each other. We collected 3750 traces from 75 different machines: 27 Apple iMac 10, 27 Dell Optiplex 750, and 21 Dell Optiplex 980; in previous trials, we were able to identify these machines with upwards of 96% accuracy. We used the dataset from one of the phones to train a Random Forest classifier to detect the Model Number attribute. When evaluated against the dataset from the second, the model achieved 100% accuracy. This test demonstrates the feasibility of using multiple USB Fingerprinting devices within an enterprise.

D. Resistance to Concept Drift

We now consider the resiliency of USB Fingerprinting against *concept drift*, the well-known machine learning problem in which a model's performance degrades over time. Concept drift would undermine our arguments regarding the adoptability of USB Fingerprinting, as ideally system administrators will be able to re-use machine learning models for months or years before re-training is required. Our study tested for concept drift by re-evaluating one of our models against new data that was collected 3 months after our original trials. Using the *Model Number* classifier described in Section IV-B, we attempted to predict the label of newly collected traces from 20 Dell Latitude e6500's and 11 Apple iMac 10.1's. These machines were selected due to ease of access, and because their computer labs had not seen much machine turnover during the 3 month span. The old *Model Number* classifier achieved 97% accuracy over the newly collected evaluation dataset. These results show that USB-based machine fingerprinting methodologies are resistant to concept drift over long spans of time, which contrasts with previous work [8] that showed inconsistent characteristics after a two-week period.

VI. Discussion

A. Towards Commodity Deployment

In an effort to encourage the further exploration of USB Fingerprinting, we are releasing the two Android applications that were developed in this study (see Section VIII). While these apps greatly simplify USB collection and analysis, we stress that our methodologies are generally employable using any device that can record USB timing events at sufficient granularity. For example, we replicated the results of Section IV-B using Gumstix, an inexpensive embedded device. We imagine a variety of potential applications for USB Fingerprinting, from attesting personal computers with our Android app, to datacenter-wide monitoring using dedicated hardware.

Android USB Analyzer: this application rapidly collects USB enumeration data from USB-enabled hosts and embedded devices. The app requires approximately 2 seconds to record and conclude a single enumeration period. In most of our experiments, we collected 50 traces from each machine. If an organization employing USB Fingerprinting used our collection procedure, this app would allow a single administrator to collect from as many as 260 host machines in a typical eight-hour day. The app does not require human intervention after initial connection to the target host, allowing the administrator to perform other tasks during data collection. The app also stores the attribute information of previously collected machines, shown in Figure 8, minimizing the amount of data entry required prior to data collection.

Android USB Identifier: this distributed application performs real-time machine identification using a previously generated fingerprint model. This model can be generated in Weka using the traces collected from *Android USB Analyzer* by following the procedures in Section IV-C. The client app sends USB enumeration traces to a remote server, building a testing set that is used to verify the previously generated machine fingerprint. The server performs the Chi-Squared test described in Section IV-C, and then notifies the client app of the result. Once the machine is labeled as either **target** or **other**, a notification message is displayed to the user. Sample notifications are also displayed in Figure 8.

B. Attacks against USB Fingerprinting

In this section, we explore how USB Fingerprinting might perform in the presence of a strong adversary. With sufficient knowledge of our approach, an attacker-controlled machine may attempt to evade detection through manipulating the behavior of its own USB stack. Let us consider the case in which the attacker has replaced a victim's machine with one that is identically specified but under attacker control, and seeks to trick the victim's USB Fingerprinting model into incorrectly identifying the device as safe for use. The attacker can attempt this feat by 1) altering the sequence or presence of descriptor requests, 2) sending invalid data to the device that violates the USB protocol, 3) launching a mimicry attack that relays USB messages from the victim's true machine, or 4) within certain constraints, altering the timing information of his messages to the device.

1) Our approach offers strong assurances against spoofed descriptor requests. This is because, in contrast to previous work [9], [11], our scheme does not rely on the presence or absence of certain descriptor sequences to identify host attributes. We demonstrate this via a proof-of-concept attack against Davis's Windows fingerprint [9], which searches for the presence of 3 `GetDescriptor (Configuration)` requests. On a dual-booted desktop, we trained a classification model on a small set of traces from two OS's: Windows 7 SP1, and a Red Hat Linux 2.6.32 kernel. We then modified the Linux kernel source, causing the *usb_new_device* function in *hub.c* to issue an additional `GetDescriptor (Configuration)` request at the end of enumeration. Running the newly built Linux kernel, we then collected a set of test traces. While Davis' scheme as described in [9] would identify the traces as belonging to a Windows host, our classifier correctly identifies the test traces as belonging to a Linux distribution.

2) If the invalid data causes enumeration to fail, the attacker is detected. If enumeration completes, the attacker may be able to trigger unexpected results, injecting confusion into the classification process. Our scheme as described does not prevent this attack; it would need to be modified to include fail-safe mechanisms in the event of unexpected host behavior. Adding a preliminary check similar to those used by Stavrou et al. and Davis would help here, as it would force the attackers inputs to conform to those of known OS behaviors [11], [9].

3) The compromised host may attempt to leverage a remote host acting as a *proxy helper* in order to mislead a USB Fingerprinting verifier, e.g., performing a *mimicry attack*. The challenges of overcoming the latency imposed by such an attack were well-defined by Li et al. [36]; the colluding hosts can trick the verifier if they are able to keep the round-trip time to the helper (T^{local}_{send},T^{local}_{recv}) and the helper's computation (T^{helper}_{comp}) within the same time bound as the local host's computation (T^{local}_{comp}). We will conservatively assume that T^{helper}_{comp} is zero because of the helper's vast computational resources. In the values of our USB enumeration feature vectors, it was a common to observe transactions within 122μs of one another, which we adopt as a time bound for USB Fingerprinting to detect an anomaly. Previous work has shown that an optimized Linux network stack has a maximum request-response rate of processing 7985 IP packets per second, a per-packet processing time of 125 μs receiving each USB transaction over the wire [37]. If the local host is bound by this value, it will require at least 250 μs to relay packets to the proxy helper ($T^{local}_{send} + T^{local}_{recv} > 250\mu$s). As this latency is well above the inter-IRQ time that is captured by many of our features, a mimicry attack will be detected.

4) The attacker can cause arbitrary message delay; if this is sufficient to mislead the classifier, the attack will go undetected. More likely, though, the attacker will need to speed up some messages or precisely control the spacing between messages, which is significantly more complicated. This would require modification of host behavior near the USB Bus Interface level. One method of reliably controlling message spacing would be to statically *replay* the correct USB enumeration. If the attacker had access to the victim's uncompromised machine, they could statically compile the full enumeration process expected by the collection apparatus, then execute it in place of allowing the typical interactions between the upper layer of the USB stack from the serial interface engine (SIE). This attack would go undetected by the *Android USB Identifier* application. One method of strengthening our scheme against this attack would be to incorporate Stavrou et al.'s method of emulating randomly selected USB peripherals [11]. This would alter the characteristics of the collection device, forcing the attacker to guess which enumeration to replay.

C. Future Work

We will undertake a variety of additional USB Fingerprinting trials, including the investigation of the impact of system load and quiescence. Although the vector in our current scheme offers over 150 features, we believe that additional contextual information can be mined from the USB protocol, both through continued passive observation as well as active analysis. These features may help to improve USB Fingerprinting results for machine identification; our preliminary results, while promising, do not offer sufficient machine coverage for general use. In spite of this, we feel that USB Fingerprinting in its current state can serve as a point of reference in a variety of applications, which we intend to explore. While we have shown that USB Fingerprinting is robust against particular forms of tampering, a full security analysis was outside the scope of this work. We intend to demonstrate that a modified version of USB Fingerprinting is fully secure against active adversaries.

VII. Related Work

A. Fingerprinting

Fingerprinting has become a popular method for device identification, and has been used to identify home electronics [38], websites [39], [40], [41], [42], [43], [44], the operating system of VMs [45], [46], and the source of phone calls [47]. The concept of fingerprinting stems from leveraging measurable signals caused by hardware imperfections in analog circuitry to uniquely identify devices. Fingerprinting has been extensively studied and used for identifying RFID smart cards [48], Ethernet cards [49] and 802.11 devices [15], [50], [51], [12], [52], as well as users [53].

Remote fingerprint techniques identify devices using only characteristics of their communication [48]. Identifying machines remotely has been a popular method, resulting in tools like *Nmap* [54] and *Xprobe* [55] that detect operating systems by examining network traffic. While effective in some cases, network fingerprints can be fooled by systems that spoof operating systems at the network layer, such as *Honeyd* [19]. Other remote schemes, such as work by Kohno et al. [56] and Jana et.al. [57], identify machines using clock skew data. However, it has been shown that TCP and ICMP timestamps can be disabled or manipulated [18], [14], [58]. Semi-persistent network data has also been used to fingerprint devices [59], [53] and browsers [60], [61]. Services to fingerprint browsers are available commercially, and are of particular interest to advertising agencies [62]. Eckersly employs similar methods of feature evaluation in his work on browser fingerprinting [13]. His highly instructive method of plotting fingerprint surprisal distribution is sadly not applicable to our feature vector due to its size and use of noisy continuous variables. Machine learning classification techniques have been also been deployed to create accurate fingerprinting schemes for user re-authentication in smartphones. Li et al. [63] used feature extraction and SVMs to recognize an individual smartphone user's finger movements.

While many remote fingerprinting methods exist, there has been little previous exploration into host identification using USB traffic through a physical connection. A recent work by Wang et al. [11] uses USB-equipped smart phones to identify host operating systems. However, their approach of reading the contents of the URB field from packets is not effective against a knowledgeable adversary, who can manipulate packet data. Butler et al. [8] employ a USB protocol analyzer to inspect the timing of bus states. In ignoring the contents and timing of USB protocol events, their approach sacrifices critical information gain, is subject to concept drift, and fails to differentiate between basic host attributes such as model number. In contrast, we demonstrate over a 10-times larger machine corpus that USB Fingerprinting is resistant to concept drift, easily distinguishes between similar machine models, and can even be used to differentiate between hosts in a set of identically specified machines. Finally, our collection mechanism is a freely available app for commodity devices, while their study relied on an expensive specialty device that requires a field expert to operate competently.

Our approach offers several benefits over existing machine classification methods. Work proposed by Desmond et al. [15], can take one hour at a minimum to collect enough data to perform classification. With our scheme, we can collect data and make a decision in a matter of minutes. Device identification work by Gerdes et al. [49] and Brik et al. [12] differentiates unique network cards of the same model, but is only applicable to network devices. Our fingerprinting technique is applicable to any device using the USB protocol. Network-based OS classification methods such as Richardson et al.'s [64] suffer from more noise than our USB approach. By using the same classifiers investigated by these authors, we distinguish between individual OSes with 100% accuracy.

B. *Compromise Detection*

Network-based (NIDS) and host-based (HIDS) intrusion detection systems [65], [66], [67], [68] can be used for compromise detection. NIDS analyze incoming and outgoing network traffic in order to determine if a system has been infected. HIDS usually refers to software that examines audit logs for suspicious activity, looking for changes in user behavior. However, an attacker with kernel control will be able to manipulate all software on the computer, including HIDS.

Garriss et al.'s trustworthy kiosk system [16] uses a smartphone to remotely verify system integrity based on trusted computing techniques. A disadvantage of this approach is the fact that a separate visual identification channel is required. Butler et al.'s Kells system [69] provides similar guarantees, but uses a USB flash drive as a remote verifier. Ensuring physical interaction with the target machine eliminates the need for a visual channel, but the approach is still susceptible to relay attacks, such as Parno's "cuckoo" attack [1].

C. *Distance Bounding*

Distance bounding protocols have been used in numerous systems, from computers to radio. Rasmussen et al. [70] demonstrate the need for fast processing speeds on any system implementing such protocols to prevent distance spoofing. Ramaswamy et al. [71] showed that processing delay within networks has become a significant concern, and an individual packet can experience increasing delays. Since our method performs the fingerprinting task over a direct physical connection, we are able to obtain more accurate timing measurements than possible over a network. VIPER [36] demonstrates software attestation with embedded systems, and also shows resilience to similar relay or *proxy* attacks. Our discussion in Section VI-B demonstrates our robustness against the attacks on distance bounding protocols, including distance hijacking [72].

VIII. Conclusion

USB Fingerprinting is a technique that can allow for the identification of unique machines in many cases, and can accurately differentiate over host attributes such as machine model and OS. We showed that a commodity smartphone is sufficient for collecting data, thus obviating the need for expensive dedicated USB analyzers to perform these operations.

This work opens an intriguing avenue for future investigation, particularly as ever more unique interfaces, potentially possessing their own characteristic stimuli and responses, come to market. This future work will involve determining whether our techniques work on technologies such as Firewire, Apple's new "Lightning" interface, and other commodity interfaces, and whether we can loosen restrictions on detaching peripherals prior to measurement. Formalizing and modeling interactions with these protocols would be of great interest to those looking to exploit and defend these interfaces and the devices that use them.

Availability

Source code for the *Android USB Analyzer* and *Android USB Identifier* apps, along with data sets of USB enumeration traces, will be made available from our lab website at `http://osiris.cs.uoregon.edu`.

Acknowledgements

We thank our shepherd, Jonathan McCune, for his guidance and the anonymous reviewers for feedback to improve the final paper. We also thank Joe Pletcher for his technical assistance and contributions of ideas, and Patrick Traynor for his valuable comments, as well as Ellisys Corporation for their generous donation of equipment to support our experiments. This work is supported in part by the US National Science Foundation under grant number CNS-1118046. Ryan Leonard and Hannah Pruse were funded in part through an NSF REU supplement.

References

[1] B. Parno, "Bootstrapping Trust in a "Trusted" Platform," in *Proceedings of the 3rd USENIX Workshop on Hot Topics in Security (HotSec'08)*, San Jose, CA, Aug. 2008, pp. 1–6.

[2] Y. Hu, A. Perrig, and D. Johnson, "Wormhole Attacks in Wireless Networks," *IEEE Journal on Selected Areas in Communications*, vol. 24, no. 2, pp. 370–380, 2006.

[3] G. P. Hancke and M. G. Kuhn, "An RFID Distance Bounding Protocol," in *Proceedings of the First International Conference on Security and Privacy for Emerging Areas in Communications Networks (SecureComm)*, Washington, DC, USA, 2005, pp. 67–73.

[4] L. Francis, G. Hancke, K. Mayes, and K. Markantonakis, "Practical NFC Peer-to-Peer Relay Attack Using Mobile Phones," *Radio Frequency Identification: Security and Privacy Issues*, pp. 35–49, 2010.

[5] J. M. McCune, A. Perrig, and M. K. Reiter, "Seeing-is-believing: Using Camera Phones for Human-verifiable Authentication," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2005, pp. 110–124.

[6] B. Gassend, D. Clarke, M. van Dijk, and S. Devadas, "Silicon physical random functions," in *CCS '02: Proceedings of the 9th ACM Conference on Computer and Communications Security*, Washington, DC, USA, 2002, pp. 148–160.

[7] TCG, *TCG Storage Architecture Core Specification*, trusted computing group ed., ser. Specification Version 1.0, Revision 0.9 – draft. Trusted Computing Group, 2007.

[8] L. Letaw, J. Pletcher, and K. Butler, "Host Identification via USB Fingerprinting," *Systematic Approaches to Digital Forensic Engineering (SADFE), 2011 IEEE Sixth International Workshop on*, pp. 1–9, May 2011.

[9] A. Davis. "Revealing Embedded Fingerprints: Deriving Intelligence from USB Stack Interactions," in *Blackhat USA 2013*. July, 2013.

[10] Gumstix, Inc. Available: https://www.gumstix.com/.

[11] Z. Wang and A. Stavrou, "Exploiting Smart-phone USB Connectivity for Fun and Profit," in *Proceedings of the 26th Annual Computer Security Applications Conference*, ser. ACSAC '10. New York, NY, USA: ACM, 2010, pp. 357–366.

[12] V. Brik, S. Banerjee, M. Gruteser, and S. Oh, "Wireless Device Identification with Radiometric Signatures," in *Proceedings of the 14th ACM International Conference on Mobile Computing and Networking (MobiCom)*. ACM, 2008, pp. 116–127.

[13] P. Eckersley, "How unique is your web browser?" in *Privacy Enhancing Technologies*. Springer, 2010, pp. 1–18.

[14] R. Pang, M. Allman, V. Paxson, and J. Lee, "The Devil and Packet Trace Anonymization," *SIGCOMM Comput. Commun. Rev.*, vol. 36, no. 1, pp. 29–38, January 2006.

[15] D. Loh, C. Y. Cho, C. P. Tan, and R. S. Lee, "Identifying Unique Devices Through Wireless Fingerprinting," in *Proceedings of the 1st ACM Conference on Wireless Network Security*, ser. WiSec '08. New York, NY, USA: ACM, 2008, pp. 46–55. [Online]. Available: http://doi.acm.org/10.1145/1352533.1352542

[16] S. Garriss, R. Cáceres, S. Berger, R. Sailer, L. van Doorn, and X. Zhang, "Trustworthy and Personalized Computing on Public Kiosks," in *Proceedings of the 6th International Conference on Mobile Systems, Applications, and Services (MobiSys '08)*, Breckenridge, CO, USA, Jun. 2008, pp. 199–210.

[17] N. Falliere, L. O. Murchu, and E. Chien, "W32. stuxnet dossier."

[18] G. Minshall, "TCPDPRIV," http://ita.ee.lbl.gov/html/contrib/tcpdpriv.html, February 05 2004 (current release).

[19] N. Provos, "A Virtual Honeypot Framework," in *Proceedings of the 13th USENIX Security Symposium*, 2004, pp. 1–14.

[20] A. Tsymbal, "The Problem of Concept Drift: Definitions and Related Work," Trinity College Dublin, Tech. Rep. TCD-CS-2004-15, 2004.

[21] Compaq, Hewlett-Packard, Intel, Microsoft, NEC, and Phillips, "Universal serial bus specification, revision 2.0," April 2000.

[22] Ellisys, "USB Explorer 200 USB 2.0 Protocol Analyzer," http://www.ellisys.com/products/usbex200/index.php, 2013.

[23] IronKey, "Ironkey," http://www.ironkey.com/en-US/resources/, 2013.

[24] S. Kondik. (2009) Cyanogenmod. [Online]. Available: http://www.cyanogenmod.org/

[25] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The weka data mining software: an update," *ACM SIGKDD Explorations Newsletter*, vol. 11, no. 1, pp. 10–18, 2009.

[26] A. Bates, R. Leonard, H. Pruse, K. Butler, and D. Lowd, "Leveraging USB to Establish Host Identity Using Commodity Devices," University of Oregon, Tech. Rep. CIS-TR-2013-12, 2013.

[27] L. Breiman, "Random Forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, 2001.

[28] J. Quinlan, *C4. 5: Programs for Machine Learning*. Morgan Kaufmann, 1993, vol. 1.

[29] Y. Freund and R. E. Schapire, "A decision-theoretic generalization of on-line learning and an application to boosting," *Journal of Computer and System Sciences*, vol. 55, pp. 119–139.

[30] R. Caruana and A. Niculescu-Mizil, "An empirical comparison of supervised learning algorithms," in *Proceedings of the 23rd international conference on Machine learning*. ACM, 2006, pp. 161–168.

[31] S. T. King, P. M. Chen, Y.-M. Wang, C. Verbowski, H. J. Wang, and J. R. Lorch, "SubVirt: Implementing Malware with Virtual Machines," *Proceedings of the 27th IEEE Symposium on Security and Privacy*, vol. 0, pp. 314–327, 2006.

[32] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the Art of Virtualization," in *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, ser. SOSP '03. New York, NY, USA: ACM, 2003, pp. 164–177. [Online]. Available: http://doi.acm.org/10.1145/945445.945462

[33] T. Ptacek, N. Lawson, and P. Ferrie, "Dont tell joanna, the virtualized rootkit is dead," *Black Hat*, 2007.

[34] J. Rutkowska, "Introducing blue pill," *The official blog of the invisiblethings. org*, vol. 22, 2006.

[35] R. Wojtczuk, "Subverting the xen hypervisor," *Black Hat USA*, vol. 2008, 2008.

[36] Y. Li, J. M. McCune, and A. Perrig, "VIPER: Verifying the Integrity of PERipherals' Firmware," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*. ACM, 2011, pp. 3–16.

[37] A. Menon and W. Zwaenepoel, "Optimizing TCP Receive Performance," in *Proceedings of the USENIX 2008 Annual Technical Conference*, 2008.

[38] S. Gupta, M. S. Reynolds, and S. N. Patel, "ElectriSense: Single-point Sensing Using EMI for Electrical Event Detection and Classification in the Home," in *Proceedings of the 12th ACM International Conference on Ubiquitous Computing*. ACM, 2010, pp. 139–148.

[39] X. Cai, X. C. Zhang, B. Joshi, and R. Johnson, "Touching from a Distance: Website Fingerprinting Attacks and Defenses," in *CCS '12: Proceedings of the 19th ACM Conference on Computer and Communications Security*, Oct. 2012.

[40] X. Gong, N. Kiyavash, and N. Borisov, "Fingerprinting Websites Using Remote Traffic Analysis," in *Proceedings of the 17th ACM Conference on Computer and Communications Security*. ACM, 2010, pp. 684–686.

[41] L. Lu, E.-C. Chang, and M. Chan, "Website Fingerprinting and Identification Using Ordered Feature Sequences," in *Computer Security, ESORICS 2010*, ser. Lecture Notes in Computer Science, B. Gritzalis, T. Dimitris, and M. Preneel, Eds. Springer Berlin Heidelberg, 2010, vol. 6345, pp. 199–214. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-15497-3_13

[42] A. Panchenko, L. Niessen, A. Zinnen, and T. Engel, "Website Fingerprinting in Onion Routing Based Anonymization Networks," in *Proceedings of the 10th Annual ACM Workshop on Privacy in the Electronic Society*, ser. WPES '11. New York, NY, USA: ACM, 2011, pp. 103–114. [Online]. Available: http://doi.acm.org/10.1145/2046556.2046570

[43] D. Herrmann, R. Wendolsky, and H. Federrath, "Website Fingerprinting: Attacking Popular Privacy Enhancing Technologies with the Multinomial Naive-Bayes Classifier," in *Proceedings of the 2009 ACM Workshop on Cloud Computing Security*, ser. CCSW '09. New York, NY, USA: ACM, 2009, pp. 31–42. [Online]. Available: http://doi.acm.org/10.1145/1655008.1655013

[44] A. Hintz, "Fingerprinting Websites Using Traffic Analysis," in *Privacy Enhancing Technologies*, ser. Lecture Notes in Computer Science, R. Dingledine and P. Syverson, Eds. Springer Berlin Heidelberg, 2003, vol. 2482, pp. 171–178. [Online]. Available: http://dx.doi.org/10.1007/3-540-36467-6_13

[45] Y. Gu, Y. Fu, A. Prakash, Z. Lin, and H. Yin, "OS-Sommelier: Memory-only Operating System Fingerprinting in the Cloud," in *Proceedings of the Third ACM Symposium on Cloud Computing*. ACM, 2012, p. 5.

[46] Owens, Rodney and Wang, Weichao, “Non-interactive OS Fingerprinting Through Memory De-duplication Technique in Virtual Machines,” in *Proceedings of the 30th IEEE International Performance Computing and Communications Conference*. IEEE, 2011, pp. 1–8.

[47] V. A. Balasubramaniyan, A. Poonawalla, M. Ahamad, M. T. Hunter, and P. Traynor, “PinDr0p: Using Single-ended Audio Features to Determine Call Provenance,” in *Proceedings of the 17th ACM conference on Computer and communications security*, ser. CCS ’10. New York, NY, USA: ACM, 2010, pp. 109–120. [Online]. Available: http://doi.acm.org/10.1145/1866307.1866320

[48] B. Danev, T. S. Heydt-Benjamin, and S. Capkun, “Physical-layer Identification of RFID Devices,” in *Proceedings of the USENIX Security Symposium*, 2009, pp. 199–214.

[49] R. M. Gerdes, T. E. Daniels, M. Mina, and S. F. Russell, “Device Identification via Analog Signal Fingerprinting: A Matched Filter Approach,” in *In Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2006.

[50] J. Franklin, D. McCoy, P. Tabriz, V. Neagoe, J. V. Randwyk, and D. Sicker, “Passive Data Link Layer 802.11 Wireless Device Driver Fingerprinting,” in *Proc. USENIX Security Symposium*, 2006.

[51] N. T. Nguyen, G. Zheng, Z. Han, and R. Zheng, “Device Fingerprinting to Enhance Wireless Security Using Nonparametric Bayesian Method,” in *Proceedings of the 30th IEEE International Conference on Computer Communications*, Apr. 2011.

[52] S. Bratus, C. Cornelius, D. Kotz, and D. Peebles, “Active Behavioral Fingerprinting of Wireless Devices,” in *Proceedings of the 1st ACM Conference on Wireless Network Security*, ser. WiSec ’08. New York, NY, USA: ACM, 2008, pp. 56–61. [Online]. Available: http://doi.acm.org/10.1145/1352533.1352543

[53] J. Pang, B. Greenstein, R. Gummadi, S. Srinivasan, and D. Wetherall, “802. 11 User Fingerprinting,” in *Proceedings of the 13th Annual ACM International Conference on Mobile Computing and Networking*, vol. 9, 2007, pp. 99–110.

[54] G. Lyon, “Nmap Free Security Scanner,” http://nmap.org/, July 16 2010 (current release).

[55] F. Yarochkin, M. Kydyraliev, and O. Arkin, “Xprobe,” http://ofirarkin.wordpress.com/xprobe/, July 29 2005 (current release).

[56] T. Kohno, A. Broido, and K. C. Claffy, “Remote Physical Device Fingerprinting,” *IEEE Trans. Dependable Secur. Comput.*, vol. 2, pp. 93–108, April 2005.

[57] S. Jana and S. K. Kasera, “On Fast and Accurate Detection of Unauthorized Wireless Access Points Using Clock Skews,” in *Proceedings of the 14th ACM International Conference on Mobile Computing and Networking*. ACM, 2008, pp. 104–115.

[58] G. Shah, A. Molina, and M. Blaze, “Keyboards and Covert Channels,” in *Proceedings of the 2006 USENIX Security Symposium*, Aug. 2006, pp. 59–75.

[59] X. Hu and Z. M. Mao, “Accurate Real-time Identification of IP Prefix Hijacking,” in *Proceedings of the 2007 IEEE Symposium on Security and Privacy*, ser. SP ‘07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 3–17.

[60] P. Eckersley, “How Unique Is Your Web Browser?” Electronic Frontier Foundation, Tech. Rep., 2009.

[61] S. Kamkar, “evercookie,” http://samy.pl/evercookie/, October 13 2010 (current release).

[62] J. Angvin and J. Valentino-Devries, “Race Is On to ‘Fingerprint’ Phones, PCs,” *Wall Street Journal*, November 30 2010.

[63] L. Li, X. Zhao, and G. Xue, “Unobservable Re-authentication for Smartphones,” in *Proceedings of the 20th Annual Network & Distributed System Security Symposium*, 2013.

[64] D. W. Richardson, S. D. Gribble, and T. Kohno, “The Limits of Automatic OS Fingerprint Generation,” in *Proceedings of the 3rd ACM workshop on Artificial Intelligence and Security*. ACM, 2010, pp. 24–34.

[65] D. E. Denning, “An Intrusion-detection Model,” *IEEE Transactions on Software Engineering*, vol. 13, no. 2, pp. 222–232, 1987.

[66] B. Mukherjee, L. Heberlein, and K. Levitt, “Network Intrusion Detection,” *IEEE Network*, vol. 8, no. 3, pp. 26–41, 1994.

[67] R. Lippmann, J. W. Haines, D. J. Fried, J. Korba, and K. Das, “The 1999 DARPA Off-line Intrusion Detection Evaluation,” *Computer Networks*, vol. 34, no. 4, pp. 579 – 595, 2000.

[68] G. H. Kim and E. H. Spafford, “The Design and Implementation of Tripwire: A File System Integrity Checker,” in *Proceedings of the 2nd ACM Conference on Computer and Communications Security*, ser. CCS ’94, Fairfax, VA, 1994, pp. 18–29.

[69] K. Butler, S. McLaughlin, and P. McDaniel, “Kells: A Protection Framework for Portable Data,” in *Proceedings of the 26th Annual Computer Security Applications Conference*, ser. ACSAC ’10. New York, NY, USA: ACM, 2010, pp. 231–240.

[70] K. B. Rasmussen and S. Capkun, “Realization of RF Distance Bounding,” in *Proceedings of the 19th USENIX Security Symposium*, 2010, pp. 389–402.

[71] R. Ramaswamy, N. Weng, and T. Wolf, “Characterizing Network Processing Delay,” in *Global Telecommunications Conference, 2004. GLOBECOM’04. IEEE*, vol. 3. IEEE, 2004, pp. 1629–1634.

[72] C. Cremers, K. Rasmussen, B. Schmidt, and S. Capkun, “Distance Hijacking Attacks on Distance Bounding Protocols,” in *Security and Privacy (SP), 2012 IEEE Symposium on*, May 2012.