

RUCKUS: A Cybersecurity Engine for Performing Autonomous Cyber-Physical System Vulnerability Discovery at Scale

Bradley Potteiger
Johns Hopkins University
Applied Physics Lab
brad.potteiger@jhuapl.edu

Daniel Cohen
Johns Hopkins University
Applied Physics Lab
daniel.cohen@jhuapl.edu

Jacob Mills
Johns Hopkins University
Applied Physics Lab
jacob.j.mills@jhuapl.edu

Paul Velez
Johns Hopkins University
Applied Physics Lab
paul.velez@jhuapl.edu

ABSTRACT

In 2016, the Cyber Grand Challenge (CGC) provided key foundations and motivations for navigating towards an autonomous cybersecurity approach. Since that time, novel strides have been made in the areas of static analysis, vulnerability discovery, patching, and exploit generation. However, a majority of these efforts have been focused on enterprise systems, leaving a gap in the Cyber-Physical System (CPS) domain. With the rise of connected infrastructure and the introduction of 5G communications, CPS are becoming more ingrained within present-day society. Due to a large amount of legacy software, and control of safety-critical actuation, CPS are and will continue to be a huge attack vector for our adversaries to remotely deploy devastating attacks against our country with low economic cost and at scale. To combat this threat, we propose the need to apply the most beneficial concepts from the CGC to create more secure and resilient CPS. In this paper, we introduce a CPS security assessment architecture RUCKUS for autonomously identifying and analyzing CPS firmware, identifying vulnerabilities, and developing exploits. Further, our approach considers how to integrate graph analytics to extrapolate findings to firmware at scale, allowing for measuring the potential widespread impact of attacks. Our architecture is demonstrated using an automotive case study, leveraging firmware from the most popular automotive and router manufacturers to assess the real-world potential impact of CPS attacks.

CCS CONCEPTS

• Security and privacy; • Computer systems organization → Embedded and cyber-physical systems;

KEYWORDS

Cyber Grand Challenge, Autonomous Vulnerability Discovery, Cyber-Physical Systems, Firmware

ACM Reference Format:

Bradley Potteiger, Jacob Mills, Daniel Cohen, and Paul Velez. 2020. RUCKUS: A Cybersecurity Engine for Performing Autonomous Cyber-Physical System Vulnerability Discovery at Scale. In *Hot Topics in the Science of Security Symposium (HotSoS)*, April 1–3, 2020, Lawrence, KS, USA. ACM, New York, NY, USA, 10 pages. <https://doi.org/10.1145/3314058.3314064>

1 INTRODUCTION

Cyber-Physical Systems (CPS) are widely becoming ingrained within the majority of safety-critical infrastructure [20]. With the tightly coupled nature of CPS between software and the surrounding physical world, adversaries no longer require access to conduct a damaging attack against our country; they can hijack vehicles, shut down power grids, and cause massive panic from remote computer terminals. Due to the asymmetric advantage provided from cyber investments compared to traditional military technologies, countries with lower resources can now become a threat to our sovereignty [36]. CPS, due to the original standalone design and standards requirements, contain a significant amount of legacy software, creating attack vectors often protected against within the enterprise domain. As such, it is now critical to ensure that we can protect our most critical infrastructure against attacks from half a world away.

In 2016, the Cyber Grand Challenge (CGC) was a first of its kind event, providing key foundations and motivations for navigating towards autonomous cybersecurity systems [21]. Competitors successfully developed systems that detected vulnerabilities, exploited competitors, and patched software, all without human intervention. This work has further led to excitement, innovations, and the development of tools over the past couple of years. As successful as the CGC event was, the environment was mostly controlled and focused on enterprise systems. When transitioning CGC technologies to the CPS domain, multiple challenges arise including a large amount of legacy software, increased connectivity, and the tightly coupled nature between the cyber and physical environments presenting a new significant attack surface.

Legacy software in CPS presents a different attack surface compared to traditional enterprise systems, focusing instead on memory corruption vulnerabilities such as buffer overflows, web application interface vulnerabilities, and communication-based vulnerabilities that allow for message spoofing and Distributed Denial of Service (DDoS) attacks [24]. Instead of prioritizing user authentication, and data integrity, CPS security prioritizes resilience, focusing instead on availability with safe and reliable operation. As such, when prioritizing vulnerabilities to patch and potential attack vectors to protect against, it is critical to combine safety and security properties.

Current cybersecurity practices focus on two stages: discovering vulnerabilities and patching those respective vulnerabilities. However, there is oftentimes a significant amount of debate about which vulnerabilities to fix first. With limited resources, this essentially makes the hardening process more of an art than a science. In this paper, we explore how to effectively objectify the criticality of vulnerabilities beyond the traditional Common Vulnerability Scoring System (CVSS) [23] and domain-specific approach while also effectively scaling and automating the vulnerability discovery process to create a more efficient and streamlined reverse engineering process of safety-critical CPS. We hypothesize that by combining a hybrid reverse engineering and automated vulnerability discovery approach, we can develop an effective cybersecurity engine that leverages the precision of human reverse engineering while assessing the occurrence of discovered vulnerabilities within the public domain. As such, metrics can be developed representing the scale of vulnerabilities, illustrating the criticality difference between a vulnerability affecting one device versus 10 million devices around the world. The contributions of our paper are as follows:

- We develop an autonomous cybersecurity engine architecture, providing a modular approach for integrating, collecting, and analyzing reverse engineering data at scale.
- We develop a firmware correlation analytic for improving the scalability of our approach.
- We illustrate the effectiveness of our approach with applications within the automotive and router domains, demonstrating real-world risk to these industries.

The rest of the paper is organized as follows: Section 2 introduces the background relating to automated vulnerability discovery approaches and the relevant attack surface of safety-critical CPS; Section 3 introduces the threat model for our paper; Section 4 describes the architecture of our autonomous vulnerability assessment engine Ruckus; Section 5 presents an implementation of our integrated firmware discovery, vulnerability discovery, and binary correlation architecture; Section 6 uses an automotive case study to illustrate the effectiveness of our approach; Section 7 presents related work; and Section 8 provides concluding remarks.

2 BACKGROUND

With the introduction of CPS such as connected and autonomous vehicles, traditionally standalone systems are now becoming significantly reliant on software infrastructure and remote communication interfaces [35]. Current automobiles include over 100 million lines of code and 50 to 70 electronic control units (ECUs), similar to the level of a F35 fighter jet [11]. With an estimation of 50 vulnerable lines for every 1,000 lines of code [6] combined with the consistent utilization of similar libraries, once a vulnerability is found in one ECU there is a high probability that exploit propagation can occur throughout the internal safety-critical network [12].

It is no longer the case that our most critical infrastructure is protected from our most dangerous adversaries. With virtually unlimited resources, time, and motivation at their disposal, it is not a question of if but when exploitation will occur. To accurately assess the potential risk to critical systems, it is important to take an attacker-centric "red team" approach where the "good guys" attempt to exploit infrastructure and find vulnerabilities before the

adversaries' "bad guys" can. Improvements in modern-day reverse engineering and vulnerability discovery tools are allowing for this red teaming process to become more effective, aiding in the process of protecting an ever more sophisticated attack landscape; however, even with this vast progress, the reverse engineering process is still highly domain-specific, requiring a large amount of highly trained specialists, time, and effort. Compared to the adversary who oftentimes can dedicate as many resources as necessary to exploit a target, the defenders do not have that luxury, having to protect every system 100% of the time. As such, it is critical to level the playing field to allow defenders to have a more user-friendly and scalable reverse engineering capability.

Modern tools developed out of the Cyber Grand Challenge were limited to predetermined and controlled scenarios, functioning on a limited set of operating systems, and devices. In the real world, a host of challenges arise dealing with factors ranging from different programming languages, compilers, architectures, memory constraints, remote connectivity, and even the native speaking language of the developers. As such, it is an extremely hard problem to develop an autonomous solution to every single possible circumstance. This problem becomes a lot easier with the introduction of a human element. By leveraging domain-specific knowledge at the start, a large amount of metadata can be collected from tools to create a baseline knowledge set. By building in analytics to focus on translating this knowledge set among platforms, architectures, and executable formats, it becomes possible to filter through a large collection of firmware and applications to identify the files of interest (overlapping libraries, applications, and open source vulnerabilities) to inject into a distributed fuzzer for a more thorough analysis.

By leveraging metadata from the human reverse engineering process with graph analytics, natural language processing, and fuzzing vulnerabilities can be discovered within CPS firmware at scale, providing the ability to assess the potential damage and prioritization of vulnerabilities to the public. To validate this hypothesis we develop a comprehensive and integrated reverse engineering pipeline that includes data collection from the human side combined with firmware correlation, vulnerability discovery, and a web crawler for measuring the impact of discovered vulnerabilities within the wild. We implement this architecture on a customized distributed cluster, leveraging both open source and proprietary firmware to assess the impact of our approach within the context of various application domains (automotive, router, mobile phones).

3 THREAT MODEL

For our threat model, we focus on legacy CPS firmware which may have memory corruption vulnerabilities such as buffer overflows. This subset is a prime example of demonstrating the ease and ability of adversaries to discover vulnerabilities within safety-critical infrastructure. By finding a memory corruption vulnerability, the goal of the attacker is to remotely exploit the target system actuation to perform a devastating action and cause chaos.

As a classical CPS, automobiles demonstrate the tightly coupled nature between software and the physical environment. An example vehicle model includes 5 main components: a sensor cluster, actuator cluster, driving controller, remote function actuator (RFA), and telematics control unit (TCU). Additionally, various communication

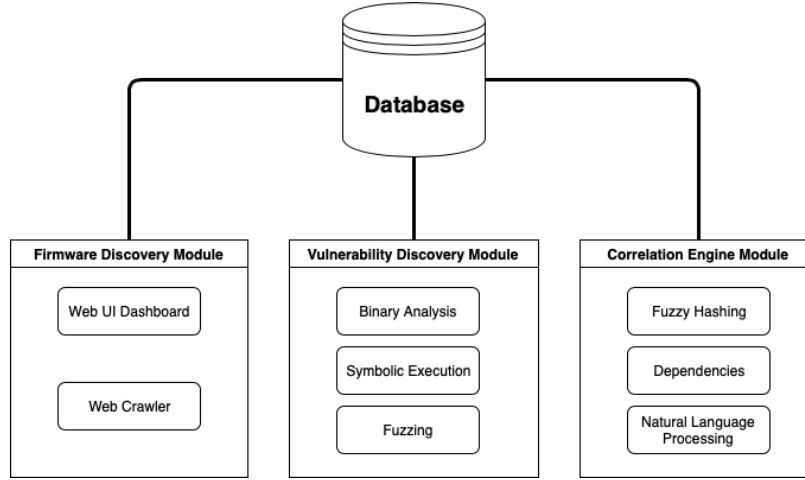


Figure 1: Ruckus Architecture Diagram

interfaces exist: an external cellular channel for communication with the central maintenance station and an internal safety-critical network for communication between the vehicle ECUs. Each ECU respectively contains a firmware image representing potential operating system vulnerabilities while the communication connections enable pivoting between vulnerable components. By being connected to public networks through the external communication channels, vulnerable components can be discovered and exploited from the Internet, making it possible for adversaries to remotely compromise safety-critical behavior.

Due to the reuse of common software, libraries, and operating system configurations, legacy vulnerabilities propagate both within CPS domains, such as within different classes of vehicles, and across other CPS domains such as a router, medical device, and HVAC firmware. In the rest of the paper, we focus on the threat of public domain vulnerabilities being utilized to compromise safety-critical components within automotive applications.

4 ARCHITECTURE

The RUCKUS automated exploitation engine is broken into three sub-modules consisting of (1) Firmware Discovery, (2) Vulnerability Discovery, and (3) Correlation, all tied together by a common database storage interface [21]. As such, our architecture is designed with this modularized approach (Figure 1).

4.1 Firmware Discovery

The first and most critical stage of our automated exploitation engine pipeline is firmware discovery. With an estimated 25 Billion IoT and embedded devices worldwide, the respective device firmware provides an avenue for identifying the software context within each environment. Since device vendors normally develop one firmware image for all of their devices in the field, analyzing the vulnerabilities, structure, and applications within this piece of firmware will allow for consequently analyzing the security of thousands of devices worldwide all at once. Since firmware is publicly available from the Internet, either through vendors or repositories,

this approach scales better compared to individually searching for and analyzing various devices on the network. Emphasizing the hybrid nature of our approach, firmware can be discovered through two avenues: by manual discovery and by autonomous discovery through a web crawler.

4.1.1 Manual Discovery: In organizations with reverse engineering departments, a queue structure is normally utilized, allocating resources to tasks based on priority, customer, and sophistication of analysis. When a device is delivered for analysis, it is not uncommon for engineers to spend multiple months to even years, analyzing every single detail of a device including hardware specifications, software applications, and vulnerabilities. As a result, an enormous amount of data is generated from this process. However, it is often the case that when another similar device is delivered from a different customer, the reverse engineering process starts from scratch, not taking advantage of a large amount of metadata generated from previous tasks. Instead of analyzing the different content of a piece of firmware, engineers spend extra time analyzing similar content, decreasing efficiency, scalability, and increasing the time to customer delivery. Therefore, in our architecture, since the manual process oftentimes generates more detailed and extensive information compared to automated scanning tools, we leverage and store this information as a foundational resource for speeding up the vulnerability discovery process.

4.1.2 Autonomous Discovery: Even though the manual reverse engineering process is useful for generating very specific and critical vulnerability information about a device, it is important to note that this approach doesn't scale well by itself. As such, we introduce an autonomous web crawler component that expands upon the set of manually inspected devices to identify similar pieces of firmware on the Internet, thus rapidly expanding the dataset. By targeting similar firmware first, we take a low-hanging-fruit approach, expanding our firmware dataset in a manner which scales based on the expertise and experience of the group, while limiting the future manual and automated analysis by leveraging previously analyzed meta-data.

4.2 Vulnerability Discovery

After accumulating target firmware, manual and automated analysis can be performed to identify various characteristics within a piece of firmware. The primary characteristics that we are focusing on include software vulnerabilities, as these usually are the most sought-after information from a customer base. Due to the necessity of scaling in our approach, we emphasize an automated analysis engine within this module.

4.2.1 Binary Analysis: The first step in an automated analysis pipeline once a piece of firmware is received is binary analysis. Binary analysis consists of extracting the file system from the firmware, iterating over and analyzing specific applications and libraries through techniques such as disassembly, control flow graph generation, and static analysis techniques. Application metadata such as symbols, functions, and linked libraries can be leveraged to further dig into the general structure, operation, and requirements of a given application.

4.2.2 Symbolic Execution. The next step after binary analysis is symbolic execution. In contrast to the binary analysis process which focuses on static information, symbolic execution takes the analysis a step further by analyzing the runtime execution of an application. By transforming a program into a sequence of mathematical equations, symbolic execution can be used to trace through a program focusing on the conditional statements and impact on control flow. Due to state explosion, symbolic execution is beneficial for a detailed analysis of specific regions of a program but is limited in terms of its scalability.

4.2.3 Fuzzing. In contrast to symbolic execution which provides a detailed analysis of the internal control flow of a program, fuzzing provides a black-box approach to analyzing the behavior of a program based on inputs. As such, by avoiding the state explosion problem, fuzzing provides our architecture with the ability to discover bugs and vulnerabilities with a high degree of scalability. However, it's difficult to analyze the full control flow path of a program. Often-times, a fuzzing approach dives in and focuses too much on the first control path that is encountered, preventing other potential vulnerable portions from being analyzed promptly. To solve this problem, we leverage a hybrid fuzzing and symbolic execution approach similar to Driller [34]. By selectively leveraging symbolic execution to guide the fuzzer to analyze paths of interest, we can increase the path of exploration while maintaining the benefits of scalability.

4.3 Correlation

The scalability behind our approach is attributed to our correlation engine. Normal reverse engineering is heavily limited depending on the number of employees, resources, and time. However, by leveraging previously discovered firmware and application metadata with discovered vulnerabilities, similar firmware can be linked together indicating the potential for similar vulnerabilities to be found. The algorithm for correlation computation is illustrated within Algorithm 1.

4.3.1 Fuzzy Hashing: Traditional firmware correlation algorithms are developed around the concept of fuzzy hashing, selectively

Algorithm 1 Compute correlation between binaries

Require: Files (F) \subseteq Binary Files (β) \subseteq {Executable, Library}
Require: Comparators (C) \subseteq {Vulns, Dependencies, Signatures, Fuzzy Hash}
Require: Target Firmware (TF) $\subseteq \beta_{TF} \subseteq C_{TF}$
Require: Dataset (D) $\subseteq \text{Firmware}_D \subseteq \beta_D \subseteq C_D$
 Matches List ML
 Binary Files BM
for all File F in TF **do**
 if F.Type $\supseteq \beta$ **then**
 Vulns_F = findVulns(F)
 Deps_F = findDeps(F)
 Sigs_F = findSigs(F)
 Hash_F = computeHash(F)
 F.comps = {Vulns_F, Deps_F, Sigs_F, Hash_F}
 BM.append(F)
 end if
end for
for all Firmware Firm in D **do**
 MatchScore score_{ba}, score_{sigs}, score_{hash}, totalscore
 counter=0
 for all File Fcur in Firm **do**
 if F.Type $\supseteq \beta$ **then**
 counter+=1
 Vulns_{Fcur} = findVulns(Fcur)
 Deps_{Fcur} = findDeps(Fcur)
 Sigs_{Fcur} = findSigs(Fcur)
 Hash_{Fcur} = computeHash(Fcur)
 score_{ba} = findOverlap(BM, Vulns_{Fcur}, Deps_{Fcur})
 score_{sigs} = findOverlap(BM, Sigs_{Fcur})
 score_{hash} = findOverlap(BM, Hash_{Fcur})
 filescore = (score_{ba} + score_{sigs} + score_{hash}) / 3
 totalscore += filescore
 end if
end for
 Match Score firmMatchScore = totalscore / counter
 Match m = {Firm_{TF}, Firm, firmMatchScore}
 ML.append(m)
end for

comparing regions of the hashes between two firmware image binary representations. This approach is beneficial for providing a quick comparison of firmware as a whole but often is limited in the ability to compare between similar applications and files. As such, we leverage fuzzy hashing at a lower level of granularity, focusing on comparing hashes between the respective applications and files within firmware images, allowing for a more fine-grained and accurate correlation.

4.3.2 Dependencies: In addition to fuzzy hashing, we can also compare firmware images concerning the various libraries, applications, and software versions within a file system. By leveraging open source vulnerability scanners we can generate CVE metadata to compare high-level vulnerability information between firmware images without the need for additional in-depth analysis.

4.3.3 Natural Language Processing: Natural Language Processing (NLP) has quickly become a popular tool for analyzing vulnerability information within systems. By analyzing the disassembly and functions of a program concerning public streams of information for databases, social network accounts, and news sites, signatures can start to be identified to indicated potential vulnerabilities. Additionally, by leveraging NLP with respect to the disassembly, applications can be compared for similar signatures, indicating a potential correlation and even vulnerability attack surfaces.

4.4 Database Storage

To leverage past reverse engineering insights for correlating potential targets of interest, it is critical to have a method of efficient storage. We use a two-pronged approach in our architecture, leveraging both a graph-based database for analyzing relationships between metadata and a cloud storage bucket for holding the actual firmware objects; these components are described in further detail below.

4.4.1 Graph Database: The foundation behind our scaling approach revolves around the correlation between various firmware objects. As such, we can leverage heuristics to cluster vulnerability analysis based on similarities between objects. To accomplish this task, we utilize a graph database that can efficiently store metadata in a manner that can be easily queried and analyzed, finding connections not easily determined by human effort.

4.4.2 Storage Bucket: Since it is not efficient to store the actual firmware objects within the graph database, we take the approach of storing direct data within a cloud storage bucket. At this point, the metadata in our graph database can remain minimal with fast lookup speeds, while including direct links to objects in bucket storage for obtaining the actual images, files, and applications.

5 IMPLEMENTATION

For the implementation of our architecture, we focus on leveraging open source tools for the majority of our components while utilizing proprietary resources for obtaining domain-specific metadata. However, by focusing on leveraging preexisting tools, we can speed up the time to deployment, as well as interact with the community to expand the breadth of our solution. We take a modular approach, adapting each tool with a Flask rest API wrapper to ease the integration with the system as a whole. The specific components are described in depth below as well as illustrated in Figure 2.

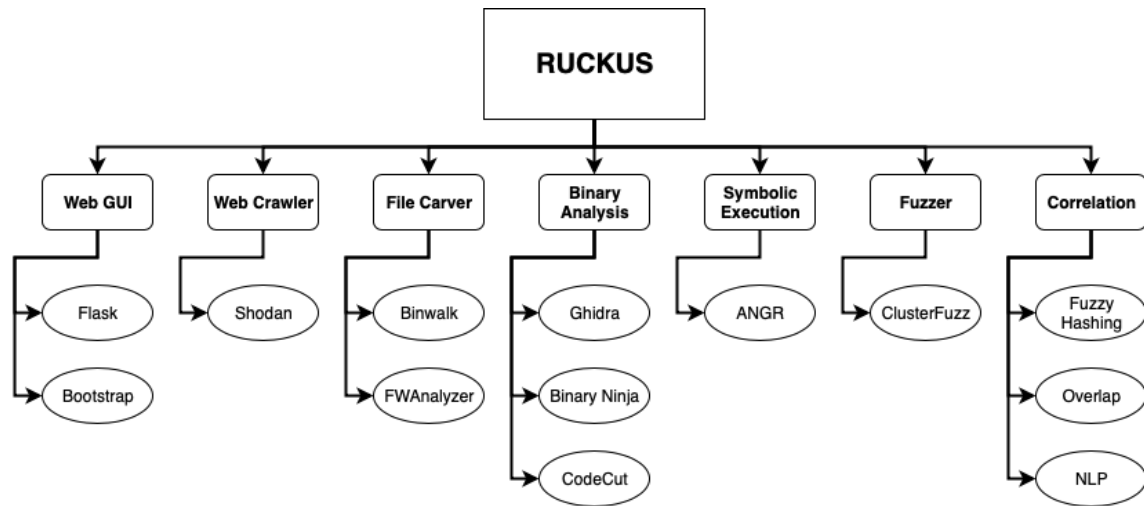
5.1 Firmware Discovery Module

The firmware discovery module is responsible for serving as the front end resource to accumulate firmware for analysis. We develop both a manual input process to leverage metadata gathered by reverse engineering teams, while also using a web crawler for scaling the dataset with public firmware. Even though the information gathered through the web crawling process is less detailed compared to the manual process, the information gathered in this process is more significant for our vulnerability process, as there is a high probability that the discovered systems can be compromised by remote exploits through the Internet. With regards to safety-critical CPS that can lead to dangerous consequences during

compromises, recognizing discovery from web crawlers can help lead to appropriate mitigations for protecting critical infrastructure. Concerning the manual firmware metadata collection process, our goal is to make the process as efficient as possible for the reverse engineering team. We developed a web-based dashboard built on Flask [4] and Bootstrap [2] to provide a streamlined and visual entry pipeline. Due to the modular architecture of Flask, we integrate our dashboard with API wrappers from common tools to develop an automated metadata extraction pipeline, significantly decreasing the amount of time and effort required to record each manual analysis process. As such, reverse engineering experts can focus on more in-depth analysis, allowing for a higher quality degree of data for our architecture. Additionally, for our web crawler, we leverage the Shodan web crawler software as a service platform which is widely regarded as the best resource for IoT device discovery [9]. Shodan not only allows for analyzing local networks, but also devices from around the world, giving us a broad and diverse dataset of various architectures, operating systems, and software versions including legacy software.

5.2 Vulnerability Discovery Module

The vulnerability discovery module is the backend responsible for performing analysis of the respective input firmware with regards to vulnerabilities, applications, libraries, and files. This module is broken down into four sub-components consisting of file carving, binary analysis, symbolic execution, and fuzzing. First, the process of file carving involves extracting the file system from the firmware image and recording high-level information such as the number and identity of various libraries, applications, misconfigurations of security protections, and potential high-level vulnerabilities allowing for remote access to the system. For high-level analysis of firmware images, we leverage FWAnalyzer [3], an open-source tool provided by Cruise Automation, which allows for analyzing any conflicts, vulnerabilities, and misconfigurations based on a given specification document. Additionally, for extracting the file system from the firmware image we leverage Binwalk [17] which is an industry-standard for reverse engineering, format analysis, and pattern recognition of binary blob objects. Our second sub-component is binary analysis. By iterating through the various applications, and libraries extracted by Binwalk, we piece together data structures of high-level binary information such as symbol names, linked libraries, functions, control flow graphs, and vulnerable instructions (e.g. strcpy, memcpy, etc.). For this purpose, we mostly leverage Ghidra [28] and CodeCut (JHU/APL Developed Software) [18], but also have the capability to integrate with Binary Ninja [1]. Third, we leverage Angr, a popular tool spun out of the Cyber Grand Challenge, for our symbolic execution capability [37]. Finally, we utilize Google's Clusterfuzz open source software for our fuzzing infrastructure [7]. Modeled after driller [34], our fuzzing infrastructure integrates traditional fuzzing (AFL) for high scalability and efficiency with symbolic execution for increasing the breadth of control paths explored for a given program.



5.3 Correlation Module

The correlation module is responsible for the scaling of our vulnerability analysis. Manual reverse engineering is time and resource-intensive, not scaling well to a high volume of applications, while a solely automated approach scales well but produces less detailed output compared to the manual process. We take a hybrid approach, attempting to correlate firmware fetched by the web crawler with previously analyzed results. For our implementation, we have three analytic sub-components consisting of Fuzzy Hashing, system overlap, and natural language processing. These three techniques provide comparison analysis at three levels of granularity, combining for a comprehensive correlation analysis of firmware images. The firmware that is identified as a high match with previously identified software will be analyzed first within our vulnerability discovery module, building up our dataset from a common knowledge base. All of our analytics are built on top of our database systems, consisting of Neo4J [5] for a graph analytics engine leveraging recorded metadata, and cloud S3 buckets for storing the firmware in full. Our fuzzy hashing approach is built on top of previous firmware correlation approaches, but moves the analysis down a level of granularity, focusing on the comparison between specific files versus the firmware image as a whole. Secondly, our overlap algorithm focuses on matching applications, libraries, CVEs, and text files with pre-identified objects within our analysis knowledge base. Finally, we developed a natural language processing algorithm to compare various signatures and function calls within executables between firmware images, to identify relationships with regards to program structure. By linking our algorithm with public sources such as CVE databases, news sources, and twitter, we can further correlate various executables and libraries with publicly disclosed vulnerabilities.

By tying in firmware discovery, vulnerability discovery, and correlation with a graph-based database, we can leverage a hybrid manual and automated vulnerability discovery approach to identify the potential propagation of discoveries within the public domain. Additionally, by tying back discoveries and correlation relationships

to our developed web-based user dashboard, we can create a more user-friendly, streamlined, and efficient reverse engineering process, giving critical information to stakeholders faster. Finally, by leveraging correlation analytics to identify the spread of our found vulnerabilities, stakeholders can quickly identify and prioritize the vulnerabilities to fix within their system (1 device affected versus 10 million, critical targets of interest, etc.).

5.4 Process Flow

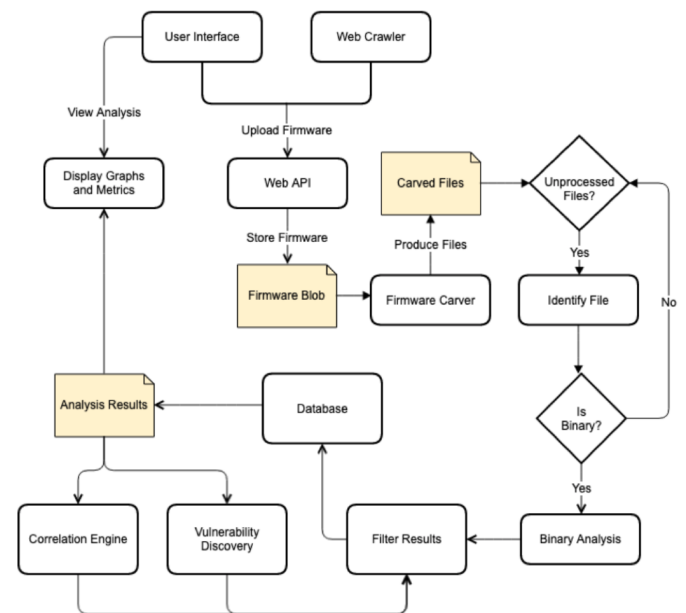


Figure 3: Process Flow

To better illustrate our implementation, we describe the process flow of our design. This process flow is illustrated in Figure 3.

The first stage of our pipeline is firmware ingestion, serving as the entry point, bringing in external firmware images into our architecture. There are two possible entry points, through manual upload, or discovery from a web crawler. At this point, the firmware image is passed to the file carver module. At this stage, firmware extraction takes place through Binwalk and FWAnalyzer, extracting the file system, and performing iteration to discover relevant files of interests including applications, libraries, and configuration files. Additionally, high-level vulnerabilities are recorded such as remote adversary entry points, and misconfigurations. At this point, the set of discovered files of interest is passed to the vulnerability discovery stage, after which a check iteration is performed to determine the subset of the discovered files that are a binary format, including popular formats such as ELF and PE. This subset is then iterated over with our binary analysis capabilities, performing symbol extraction, static analysis and symbolic execution using Ghidra and Angr to extract high-level descriptive metadata about the overall behavior and semantics of the target binary. Then, the extracted meta-data is filtered to descriptors of interest and input into the graph database Neo4j. Additionally, the actual binary object is uploaded to our bulk storage bucket for access at a later date.

In parallel to this analysis process, our vulnerability and correlation analysis modules iterate over respective data within our databases. The vulnerability analysis module iterates over nodes within Neo4j to discover any binaries not previously analyzed for vulnerabilities. The relevant metadata is then taken as input, including the actual binary file of interest. With symbols, static analysis, and control flow meta-data as input, we then feed this information into our analysis and fuzzing engine to discover potential vulnerabilities. Additionally, the binary file node is analyzed within the Neo4j database to find similar files with discovered vulnerabilities. This information is then checked against the current binary file and included within the vulnerability output report along with the newly discovered vulnerabilities. Finally, the target binary file node in Neo4j is updated with the respective vulnerability meta-data. The correlation module iterates through binary file nodes within the Neo4j database to analyze connections between previously analyzed binary files. By utilizing techniques such as fuzzy hashing, metadata overlap, and natural language processing, a match score is developed to represent the similarity of the target binary file to other binary files within our dataset. This matching score can then be leveraged as a heuristic in the vulnerability discovery process of binary files in the future.

To complete the process flow of our pipeline, the discovered metadata results are fed back into our web-based user interface for visualization. Our user interface dashboard also provides the ability to perform additional queries on the binary file dataset to further explore the gathered metadata within our firmware collection.

6 EVALUATION

As mentioned throughout the previous sections of this paper, an automated vulnerability discovery and exploitation engine is beneficial for defenders in identifying and mitigating vulnerable devices within their network. However, this class of tool is equally as beneficial for adversaries to scan and find exploitable targets before mitigations are implemented. Furthermore, this can be especially

detrimental in cases where open source vulnerabilities can be leveraged to exploit proprietary CPS software that is often outdated and relies on security through obscurity approaches [25]. To address the possibility of an attacker leveraging vulnerabilities from open source software to exploit critical infrastructure, we make use of our automated vulnerability discovery tool RUCKUS to provide insights.

Future models of vehicles such as automated and connected vehicles make use of increasingly connected and distributed architectures while still maintaining the original underlying codebase through decades of designs. As such, instead of redesigning software from scratch, the trend is to build on top of the existing codebase, leaving a significant portion of legacy software underneath. With the introduction of remote communication interfaces, these systems are easily susceptible to exploits not originally taken into account when designing the system. By remotely exploiting a vehicle, potentially from another country, adversaries can leverage the actuation capabilities to inflict real, devastating damage.

The primary defense mechanism of automotive software is security through obscurity [25]. Vehicle firmware is often proprietary and hard to acquire for analysis. Manufacturers hope that this lack of public knowledge about specific software applications and protocols will prevent the reconnaissance knowledge necessary to exploit their systems. Traditional approaches in-vehicle security research have assumed that attackers would have access to this underlying firmware to discover potentially exploitable vulnerabilities. However, in this case, we want to take the approach that the firmware is unobtainable by the attacker. In this scenario, adversaries generally leverage previously obtainable reconnaissance information from other similar classes of devices to predict the most likely vulnerabilities and exploits.

With publicly accessible firmware from routers, programmable logic controllers (PLCs), and SCADA systems, it is reasonable that vulnerabilities from these classes of devices can be found in automobiles as well. In this case study, we illustrate the use of our automated vulnerability discovery tool RUCKUS to gain insight into the similarities between open-source router firmware and proprietary automobile firmware.

6.1 Attack Scenario

In our attack scenario, we assume that the adversary doesn't have access to the respective automobile firmware to analyze. The adversary intends to leverage open-source information from a similar class of devices such as a router. By assessing the occurrence of these vulnerabilities within the most popular manufacturers and leveraging the monolithic software design of the vehicle makes and models within a respective manufacturer, the adversary aims to propagate the exploit throughout the public, affecting as many vehicles as possible.

6.2 Dataset

To accurately assess the similarities among firmware, software libraries, and vulnerabilities we make use of both open-source router firmware and proprietary automotive firmware. For the router firmware, we make use of 25 firmware images of the most popular manufacturers including Belkin, Cisco, DD-WRT, Netgear,

and Linksys. To include a comprehensive assessment of potential software library versions, we leverage a combination of firmware images from the past 10 years.

For the automotive dataset, we leverage 5 proprietary firmware images through the vehicle analysis group at the Johns Hopkins University Applied Physics Laboratory (JHU/APL). To protect the community, we are not releasing the manufacturers, makes, or models of this firmware. The number of vehicles on the road today making use of the firmware from these manufacturers is estimated to exceed 10 million, making any compromise of this software a devastating and wide-scale attack.

6.3 Results

To assess the similarities between the different firmware images, we illustrate the process of RUCKUS while providing results from each step. In the first stage, we input the firmware images into our system through a web-based dashboard provided to our reverse engineering team. This front end was built in a manner making it easy to drag and upload a bulk number of images to perform large scale analysis. The front end web application is illustrated in Figure 4.

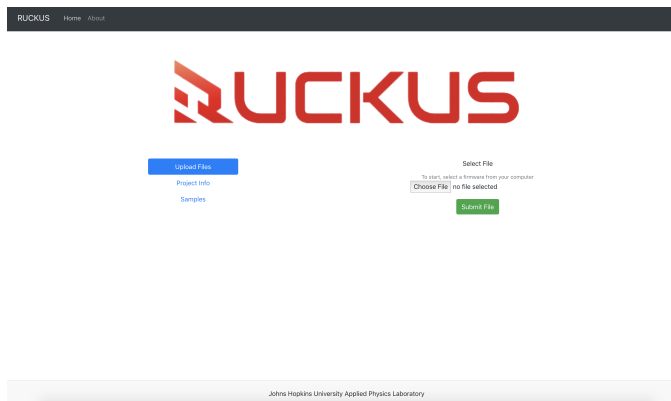


Figure 4: Web App Front End

After uploading, the router firmware images are extracted with Binwalk and parsed to identify files of interest such as shared libraries, executables, and configuration files. The results of this process are illustrated in Figure 5. As can be observed, each piece of firmware averages approximately 221 files of interest, including 129 shared library files, 80 executables, and 12 configuration files.

After firmware extraction, each file of interest, as well as the file system as a whole, is analyzed for vulnerabilities including static analysis inspection through Ghidra, open-source CVE discovery through a natural language processing module, and runtime vulnerability inspection through our fuzzing engine. This process produced an average of 15 potential vulnerabilities per firmware image. These results are illustrated in Figure 6.

After vulnerability discovery, firmware metadata is entered into the Neo4j graph database accompanied by relevant found vulnerability information, such as affected library names, and versions, type of vulnerabilities, quantity, and ease of replication. This information forms the foundation for the firmware similarity comparison

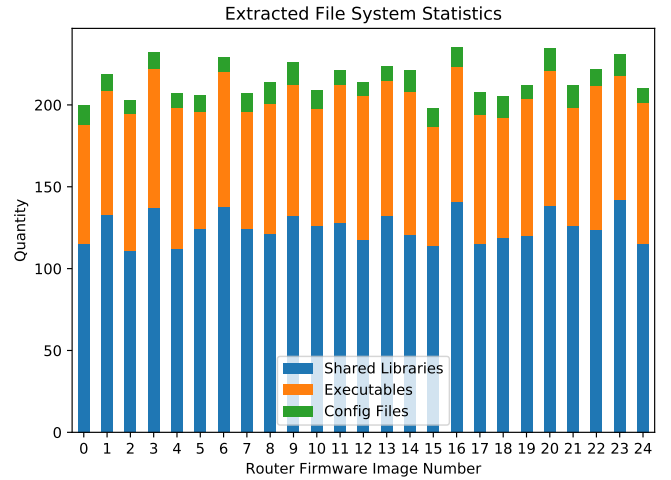


Figure 5: Extracted File System Statistics

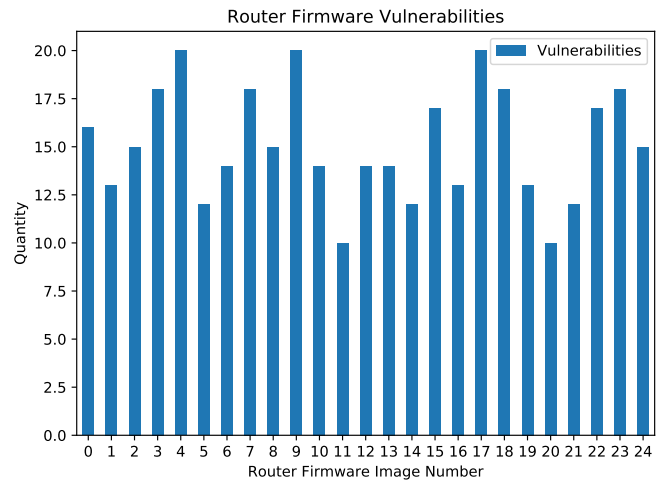


Figure 6: Router Firmware Discovered Vulnerabilities

process. At this stage, the first and second steps are repeated with the automotive firmware, uploading and extracting the firmware images. Instead of directly performing vulnerability discovery on the images, we input the extracted firmware metadata through our firmware similarity comparison module. The automotive firmware is compared to the router firmware with regards to similar shared libraries, executables, and configurations, consequently producing a match score between 0 and 1. The higher the match score the more similarity that exists between firmware images. The results show that there is a fairly high match score (max score: 62%) between the proprietary automotive firmware and open-source router firmware indicating that there are several common libraries and applications used within the software. Additionally, higher match scores are found with older firmware images indicating that several of the automotive firmware relies on outdated software libraries.

The match scores between the automotive and router firmware are illustrated in Figure 7.

Firmware Matching Scores (x100)					
Router ID	Auto1	Auto2	Auto3	Auto4	Auto5
Cisco '17	49	32	31	29	21
Belkin '16	51	38	29	36	24
Linksys '17	46	43	38	39	22
DD-WRT '19	52	32	36	44	29
Cisco '16	48	48	41	45	48
Belkin '15	58	60	46	45	41
Linksys '16	59	51	54	58	39
DD-WRT '06	62	58	53	49	51
DD-WRT '08	55	53	56	53	53
Belkin '14	49	51	47	51	50
DD-WRT '13	50	53	51	48	49
DD-WRT '17	48	48	47	48	48
Linksys '18	47	51	44	45	43
DD-WRT '18	42	44	43	42	44
Netgear '10	41	40	41	42	41
Netgear '12	36	35	38	41	30
Netgear '14	42	37	36	32	31
DD-WRT '20	31	34	39	31	31
Linksys '19	29	31	30	29	29
Netgear '16	23	22	26	27	23
Belkin '18	25	20	26	23	21
Cisco '18	12	21	24	18	20
Netgear '18	20	16	15	14	12
Netgear '19	23	21	20	11	14
Netgear '20	18	14	15	16	17

Figure 7: Match Scores

The final stage is rapid vulnerability translation. By starting with the firmware images with the highest match scores, the previously discovered vulnerabilities for the respective shared libraries and executables are assigned to their matches within the automotive firmware. At this point, the reverse engineering teams have an initial understanding of the most probable vulnerabilities within the automotive firmware images. For this case study, the vulnerabilities were validated to be true within the automotive firmware images. The results of the discovered automotive vulnerabilities are illustrated in Figure 8. It is important to note that these vulnerabilities are not extremely sophisticated zero days but are often very simple memory corruption vulnerabilities such as buffer overflows that allow for remote exploitation of a system. This indicates and validates our hypothesis that the legacy foundation of these systems is often providing the most risk to the attack surface.

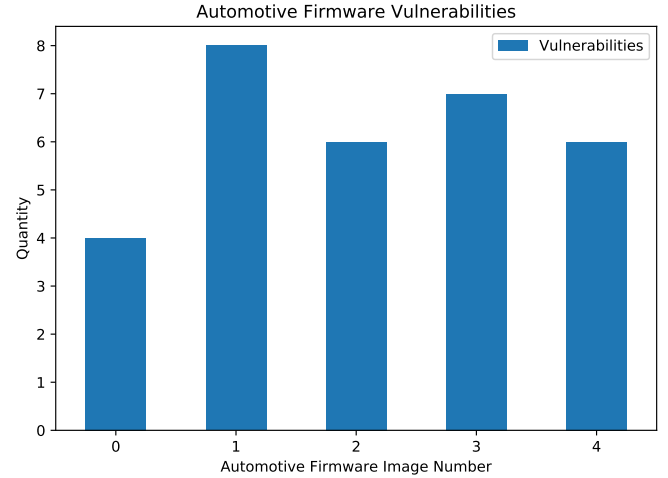


Figure 8: Automotive Vulnerabilities

7 RELATED WORK

Output from the Cyber Grand Challenge [30, 31] consisted of static analysis [37], binary analysis [28], fuzzing [27, 34], patching [15, 19], and ricochet attack tools [8], even providing the foundation for commercialization [10, 16]. However, this competition was limited to a customized operating system called DECREE (DARPA Experimental Cyber Research Evaluation Environment) [21]. Even though this framework provided a control scenario for accurately judging competitors, real-life scenarios are much messier and complicated.

Since 2016, efforts have been made to transition the output technology to tools compatible with working in the real world. With regards to fuzzing, there has been more intensive research on an integrated symbolic execution, AFL approach [13, 26, 39] to optimize performance and maximize the diversification of control flow paths, including the commercialization into wide-scale products such as Google's Clusterfuzz [7]. In terms of static analysis and vulnerability assessment ANGR [37] has been significantly improved upon while the release of Ghidra [28] has lowered the barrier of entry to rapidly improve upon binary analysis techniques. Additionally, significant research interest has been given to binary rewriting tools and assured micro-patching techniques including the start of the DARPA Assured Micropatching Program [14], development of Gramma Tech's GTIRB [29], Trail of Bits' McSema [15], and Columbia University's Egalito platforms [38]. Finally, there have been significant improvements in firmware analysis and crawling capabilities including correlation efforts from Firmware.RE [40], fuzzing capabilities provided from FirmFuzz [32] and extraction and analysis capabilities provided from Firmadyne [22].

8 CONCLUSION

In this paper, we have introduced our autonomous embedded vulnerability discovery engine RUCKUS. RUCKUS leverages manual fine-grained inspection, combined with autonomous web crawling, correlation, and fuzzy hashing for analyzing vulnerabilities of embedded firmware at scale. We introduced the stages of RUCKUS

which are firmware discovery, firmware extraction, vulnerability discovery, correlation, and database storage. Finally, we developed a case study to illustrate the process flow of our tool, focusing on analyzing the similarities between open-source router firmware and proprietary automotive firmware to assess the ability of attackers to exploit common vulnerabilities. In the future, we plan to further refine our correlation engine, explore natural language processing for identifying CVEs in firmware, and upgrade our web crawling and fuzzing capabilities to assess the scalability of our approach. Additionally, we plan to integrate the AllStar dataset for increasing the quantity and diversity of firmware in our knowledge base [33].

REFERENCES

- [1] Binary ninja. Available at <https://binary.ninja>.
- [2] Bootstrap for web application design. Available at <https://getbootstrap.com>.
- [3] Cruise automation fwalyzer. Available at <https://github.com/cruise-automation/fwalyzer>.
- [4] Flask python library. Available at <https://www.palletsprojects.com/p/flask/>.
- [5] Neo4j graph database. Available at <https://neo4j.com>.
- [6] Percentage of vulnerabilities per lines of code. Available at <https://www.hackerone.com/blog/What-percentage-your-software-vulnerabilities-have-GDPR-implications>.
- [7] A. Arya, O. Chang, M. Moroz, M. Barbella, and J. Metzner. The clusterfuzz team. 2019. open sourcing clusterfuzz. google open source blog.
- [8] T. Bao, Y. Shoshitaishvili, R. Wang, C. Kruegel, G. Vigna, and D. Brumley. How shall we play a game?: A game-theoretical model for cyber-warfare games. In *2017 IEEE 30th Computer Security Foundations Symposium (CSF)*, pages 7–21. IEEE, 2017.
- [9] R. Bodenheimer, J. Butts, S. Dunlap, and B. Mullins. Evaluation of the ability of the shodan search engine to identify internet-facing industrial control devices. *International Journal of Critical Infrastructure Protection*, 7(2):114–123, 2014.
- [10] D. Brumley. For all secure. Available at <http://www.forallsecure.com>.
- [11] R. N. Charette. This car runs on code. *IEEE spectrum*, 46(3):3, 2009.
- [12] S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, S. Savage, K. Koscher, A. Czeskis, F. Roesner, T. Kohno, et al. Comprehensive experimental analyses of automotive attack surfaces. In *USENIX Security Symposium*. San Francisco, 2011.
- [13] P. Chen and H. Chen. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 711–725. IEEE, 2018.
- [14] DARPA. Assured micropatching program. Available at <https://www.darpa.mil/news-events/2019-10-14>.
- [15] A. Dinaburg and A. Ruef. Mcsema: Static translation of x86 instructions to llvm. In *ReCon 2014 Conference, Montreal, Canada*, 2014.
- [16] D. Guido. Trail of bits. Available at <http://www.trailofbits.com>.
- [17] C. Heffner. Binwalk: Firmware analysis tool. URL: <https://code.google.com/p/binwalk/visited> on 03/03/2013, 2010.
- [18] JHU/APL. Github. Available at <https://github.com/JHUAPL/CodeCut>.
- [19] A. H. S. N. P. Larsen and S. B. M. Franz. Profile-guided automated software diversity.
- [20] E. A. Lee. Cyber physical systems: Design challenges. In *2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*, pages 363–369. IEEE, 2008.
- [21] N. Lee. Darpa’s cyber grand challenge (2014–2016). In *Counterterrorism and Cybersecurity*, pages 429–456. Springer, 2015.
- [22] D. Liu, Y. Tang, B. Wang, W. Xie, and B. Yu. Automated vulnerability detection in embedded devices. In *IFIP International Conference on Digital Forensics*, pages 313–329. Springer, 2018.
- [23] P. Mell, K. Scarfone, and S. Romanosky. Common vulnerability scoring system. *IEEE Security & Privacy*, 4(6):85–89, 2006.
- [24] C. Miller and C. Valasek. Remote exploitation of an unaltered passenger vehicle. *Black Hat USA*, 2015, 2015.
- [25] S. Mohan, S. Bak, E. Betti, H. Yun, L. Sha, and M. Caccamo. S3a: secure system simplex architecture for enhanced security of cyber-physical systems. *arXiv preprint arXiv:1202.5722*, 2012.
- [26] S. Ognawala, T. Hutzelmann, E. Psallida, and A. Pretschner. Improving function coverage with munch: a hybrid fuzzing and directed symbolic execution approach. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, pages 1475–1482. ACM, 2018.
- [27] H. Peng, Y. Shoshitaishvili, and M. Payer. T-fuzz: fuzzing by program transformation. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 697–710. IEEE, 2018.
- [28] R. Rohleder. Hands-on ghidra-a tutorial about the software reverse engineering framework. In *Proceedings of the 3rd ACM Workshop on Software Protection*, pages 77–78, 2019.
- [29] E. Schulte, J. Dorn, A. Flores-Montoya, A. Ballman, and T. Johnson. Gtibr: Intermediate representation for binaries. *arXiv preprint arXiv:1907.02859*, 2019.
- [30] J. Song and J. Alves-Foss. The darpa cyber grand challenge: A competitor’s perspective. *IEEE Security & Privacy*, 13(6):72–76, 2015.
- [31] J. Song and J. Alves-Foss. The darpa cyber grand challenge: A competitor’s perspective, part 2. *IEEE Security & Privacy*, 14(1):76–81, 2016.
- [32] P. Srivastava, H. Peng, J. Li, H. Okhravi, H. Shrobe, and M. Payer. Firmfuzz: Automated iot firmware introspection and analysis. In *Proceedings of the 2nd International ACM Workshop on Security and Privacy for the Internet-of-Things*, pages 15–21, 2019.
- [33] J. Staff. Assembled labeled library for static analysis research (allstar) dataset, Dec 2019.
- [34] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS*, volume 16, pages 1–16, 2016.
- [35] I. Studnia, V. Nicomette, E. Alata, Y. Deswarte, M. Ka  n  che, and Y. Laarouchi. Survey on security threats and protection mechanisms in embedded automotive networks. In *Dependable Systems and Networks Workshop (DSN-W), 2013 43rd Annual IEEE/IFIP Conference on*, pages 1–12. IEEE, 2013.
- [36] R. Syed, A. A. Khaver, and M. Yasin. Cyber security: Where does pakistan stand? 2019.
- [37] F. Wang and Y. Shoshitaishvili. Angr-the next generation of binary analysis. In *2017 IEEE Cybersecurity Development (SecDev)*, pages 8–9. IEEE, 2017.
- [38] D. Williams-King. Egalito. In *ASPLOS 2020*.
- [39] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim. {QSYM}: A practical concolic execution engine tailored for hybrid fuzzing. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 745–761, 2018.
- [40] J. Zaddach and A. Costin. Embedded devices security and firmware reverse engineering. *Black-Hat USA*, 2013.