FINGERPRINTING VULNERABILITIES IN INTELLIGENT ELECTRONIC DEVICE FIRMWARE

Leo Collard

A thesis
in
The Department
of
Concordia Institute for Information Systems Engineering
(CIISE)

Presented in Partial Fulfillment of the Requirements
For the Degree of Masters of Applied Science in Information Systems Security
Concordia University
Montréal, Québec, Canada

December 2018

CONCORDIA UNIVERSITY
School of Graduate Studies

This is to certify that the thesis prepared

By:	**Leo Collard**
Entitled:	**Fingerprinting Vulnerabilities in Intelligent Electronic Device Firmware**

and submitted in partial fulfillment of the requirements for the degree of

Masters of Applied Science in Information Systems Security

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining commitee:

Dr. C. Wang	______________________	Chair
Dr. M. Debbabi	______________________	Supervisor
Dr. A. Hanna	______________________	Co-supervisor
Dr. A. Youssef	______________________	CIISE Examiner
Dr. J. Paquet	______________________	External Examiner (CSE)

Approved ______________________________________
Chair of Department or Graduate Program Director

______________ 20 ______ ______________________________

Dr. Amir Asif, Dean
Gina Cody School of Engineering and Computer Science

Abstract

Fingerprinting Vulnerabilities in Intelligent Electronic Device Firmware

Leo Collard

Modern smart grid deployments heavily rely on the advanced capabilities that Intelligent Electronic Devices (IEDs) provide. Furthermore, these devices firmware often contain critical vulnerabilities that if exploited could cause large impacts on national economic security, and national safety. As such, a scalable domain specific approach is required in order to assess the security of IED firmware. In order to resolve this lack of an appropriate methodology, we present a scalable vulnerable function identification framework. It is specifically designed to analyze IED firmware and binaries that employ the ARM CPU architecture. Its core functionality revolves around a multi-stage detection methodology that is specifically designed to resolve the lack of specialization that limits other general-purpose approaches. This is achieved by compiling an extensive database of IED specific vulnerabilities and domain specific firmware that is evaluated. Its analysis approach is composed of three stages that leverage function syntactic, semantic, structural and statistical features in order to identify vulnerabilities. As such it (i) first filters out dissimilar functions based on a group of heterogeneous features, (ii) it then further filters out dissimilar functions based on their execution paths, and (iii) it finally identifies candidate functions based on fuzzy graph matching . In order to validate our methodologies capabilities, it is implemented as a binary analysis framework entitled BinArm. The resulting algorithm is then put through a rigorous set of evaluations that demonstrate its capabilities. These include the capability to identify vulnerabilities within a given IED firmware image with a total accuracy of 0.92.

Acknowledgments

I would like to deeply thank all those who have made this work possible. This starts foremost with my supervisors Dr. Mourad Debbabi and Dr. Aiman Hanna who have generously supported me, both with their time and knowledge, over the course of this degree. I would like to thank Paria Shirani who guided me through this entire process, which is very new to me. I also want to thank my lab colleagues Mark Karanfil, and Chantale Robillard who have very generously assisted me in furthering my understanding of the smart grid and the threats that it faces. They have done so by providing me with reading materials and references to pertinent research. I also want to thank Anthony Akary for the work he provided within his internship report that discussed the IEC 61850 standard. I want to thank Hydro-Québec and Thales along with Dr. Marthe Kassouf, Dr. Basile L. Agba, and Bernard Lebel for their partnership with Concordia without which this entire degree would not have been possible. Finally, I also thank all members of our weekly smart grid research meetings who provided invaluable suggestions, comments, and feedback all throughout my time at Concordia.

Contents

List of Figures

List of Tables

List of Acronyms

ACFG	Attributed CFG
ACSI	Abstract Communication Service Interface
APT	Advanced Persistent Threat
AST	Abstract Syntax Tree
BFS	Breadth First Search
BN	Bayesian Network
CDF	Cumulative Distribution Function
CEC	Concrete Executor Client
cFCG	Colored Function Call Graph
CFG	Control Flow Graph
CIP	Critical Infrastructure Protection
CIT	Conventional Instrument Transformer
CT	Current Transformers
CVE	Common Vulnerabilities and Exposures
DA	of Data Attributes
DAG	Directed Acyclic Graph
DBA	Dynamic Binary Analysis
DER	Distributed Energy Resource
DFG	Data Flow Graph
DFS	Depth First Search
DG	Dependence Graph
DOE	Department of Energy

EPRI	Power Research Institute
FCG	Function Call Graph
GED	Graph Edit Distance
GES	Evidence of Similarity
GOOSE	Oriented Substation Event
HMAC	Message Authentication Code
HMIs	Human Machine Interfaces
HMM	Hidden Markov Model
iCFG	Information Control Flow Graph
ICS	Industrial Control System
IEC	International Electrotechnical Commission
IED	Intelligent Electronic Device
IR	Intermediate Representation
LCS	Longest Common Subsequence
LD	Logical Device
LES	Local Evidence Scores
LRU	Least Recently Used
MCS	Maximum Common Subgraph
MI	Mutual Information
MMS	Manufacturing Messaging Specification
mRFG	Merged Register Flow Graph
NCIT	Conventional Instrument Transformer
NERC	Electric Reliability Corporation
NESCOR	Cybersecurity Organization Resource
NSM	Network and System Management
NTD	Normalized Tree Distance
NTP	Network Time Protocol

PDB	Program Debug Database
PDC	Phasor Data Concentrator
PDF	Probability Density Function
PDG	Program Dependency Graph
PMU	Phasor Measurement Unit
RFG	Register Flow Graph
RTU	Remote Terminal Units
SAMU	Alone Merging Unit
SAS	Substation Automation System
SCADA	and Data Acquisition
SCL	Substation Communication Language
SES	Symbolic Executor Server
SIG	Semantic Integrated Graph
SNMP	Simple Network Management Protocol
SVM	Support Vector Machine
VT	Voltage Transformers
WAN	Wide Area Networks
WJ	Weighted Jaccard Similarity
WNTD	Normalized Tree Distance

Chapter 1

Introduction

1.1 Motivations

Industrial Control Systems (ICS) are widely used in today's industries as well in critical infrastructures. These interconnected devices are powerful controllers that enable a wide range of functionality such as data acquisition, automation and control. They are composed of two main parts, namely hardware and software (firmware). One critical infrastructure that heavily relies on ICS equipment is the electrical infrastructure or more precisely the smart grid. In this context, ICS equipment plays a large role in the form of numerable devices such as Intelligent Electronic Devices (IEDs), or Remote Terminal Units (RTUs). By specifically targeting these critical devices, malicious entities are able to cause devastating effects. These include disruption of energy generation, transmission and distribution, which drastically impacts national economic security and national safety [1].

Such attacks are not merely theoretical; already several highly disruptive ones have taken place. These attacks distinguish themselves from traditional cyber attacks by having the specific intent to damage, destroy or disrupt physical systems. These assaults are thus appropriately labeled cyber-physical attacks. Stuxnet [2, 3] is probably the first known cyber-warfare weapon that is specifically designed to perform a completely standalone cyber-physical attack. It is designed to specifically target certain Siemens ICS equipment that is deployed in over fifteen Iranian nuclear facilities. This resulted in an estimated destruction of 984 uranium enrichment centrifuges, which caused a large decrease in enrichment efficiency [3]. In December 2015, the Black Energy [4] Advanced Persistent Threat (APT) took control of power grid operators control stations through a valid remote access application and utilized them to turn off power. This cyber-physical security incident resulted in $225,000$ people loosing power in western Ukraine [5]. A more recent and relevant attack is Industroyer [6, 7] which took place in December 2016 and once again targeted Ukraine [8]. This attack is the first to specifically target a power grid or more precisely a transmission substation. It is a framework that is used to facilitate the creation of attacks whose purpose is to cause power outages.

Through these attacks and many others, it is possible to observe a large upward trend over time. In fact, the number of attacks against ICS equipment has increased by 110% between 2015 and 2016 [9]. Such trends and attacks are taken very seriously by the industry. This can be observed through reports such as the one published by Lloyd's entitled Business Blackout [10]. In this article, they explore a scenario where 50 power generators in the United States are knocked out by hackers that left 93 million people without power.

1.2 Objectives

These real-world events and hypothetical scenarios indicate a clear potential for future attacks against the electrical infrastructure. These cyber-physical attacks may have immense impacts which need to be prevented, detected, mitigated and attributed. In order to achieve this goal, our primary objectives are:

- Conduct a comparative study of the state-of-the-art research proposals in the area of IED security
- Design and implement a scalable and efficient technique for the automatic analysis of IED firmware
- Create a database of actual IED firmware images and vulnerabilities
- Conduct a thorough evaluation of the designed and implemented analysis framework

1.3 Approach

Due to these threats, it is clearly essential to develop methodologies and frameworks that can perform security analyses of IEDs. However, this is no simple task due to the complexity of smart grid deployments. Previous research has demonstrated that the majority of vulnerabilities identified in ICS equipment reside in software implementations (92.6%) rather than hardware (7.4%) [11]. It is also the case that many embedded intelligent devices utilize the ARM architecture [11, 12, 13, 14]. As such, these firmware-based vulnerabilities are most likely the ones that will be exploited to perform devastating cyber-physical attacks. Performing a security analysis of these is arduous. It is almost always the case that firmware source code is not publicly available. Reverse engineering is a necessary, albeit rather challenging task [11]. This is due in part to obfuscation [15], unknown firmware formats [16] and unknown entry points [17].

The intention of this work is thus to create a methodology for systematically identifying known vulnerabilities in smart grid IED-related firmware. This is in contrast to other existing general-purpose techniques (e.g., [18, 19, 20]) that lack a focus on the smart grid context. Although usable, they suffer from some distinct limitations.

Applicability: This lack of domain knowledge implies that other methodologies cannot be easily applied to such a specialized context. Lacking aspects include databases of IED firmware images and vulnerabilities. Acquiring such information in advance facilitates the analysis process by: (i) removing the need for prior domain knowledge, (ii) identifying the most relevant vulnerabilities, and (iii) providing a concrete set of vulnerabilities to test for.

Scalability: Several methodologies leverage computationally complex comparison operations (e.g., semantic hashing [21]) and do not leverage any pre-filtration to increase the overall performance time. As such, scalability and efficiency are major issues that limit the large-scale analysis of real world smart grid IED firmware images. Furthermore, these firmware images are often quite large when compared to simple network routers.

Adaptability: The capacity to quickly and easily index new vulnerable functions is extremely important in this context. This is difficult for some methodologies (e.g., [19]).

With the purpose of surpassing these limitations, we leverage a multi-stage detection approach. The core of this process is centered around performing several phases of fuzzy comparison between a given target firmware function and a set of vulnerable reference functions using their Control Flow Graphs (CFGs) and other information derived from them. The purpose of each phase is to preserve only the most similar matches in order to accelerate the overall comparison time. The first stage identifies similar functions based on their shape [22] by calculating the Euclidean distance between two 3-tuples of light statistical features. The resulting candidate functions are then compared based on their execution paths by utilizing the Normalized Tree Distance (NTD) [23]. Finally, the relatively few remaining functions CFGs are compared using the Longest Common Subsequence (LCS) [24] along with the Hungarian algorithm [25].

In order to perform and validate such an approach, it is necessary to create three large-scale datasets. The first is a set of known vulnerable functions that are likely present inside of smart grid IED firmware. This is achieved by identifying relevant and large smart grid manufacturers. It is then possible to use this information to find libraries that are publicly accessible and commonly reused by IED firmware. From there, it is necessary to extract and classify those functions which have been reported to contain vulnerabilities [26], and store them in our vulnerability database. The second leverages the previously identified manufacturers to identify publicly available IED firmware images. These are acquired, extracted, and stored in our firmware database that is later leveraged for analyses. The third dataset is composed of all library functions that are not vulnerable. It is utilized to perform all large-scale evaluations.

1.4 Contributions

We can list the main contributions of this work as follows:

- To the best of our knowledge, we produce the first large-scale vulnerability database that is specifically tailored for IED device firmware. It covers all major manufacturers (ex. Siemens, ABB, Schneider, SEL, etc.). At the same time, we create the first IED firmware dataset that is specifically related to the smart grid.
- We propose a multi-stage vulnerability detection framework entitled BinArm. This approach outperforms existing fuzzy matching approaches with respect to speed, while maintaining accuracy. It also addresses the scalability issues of previous work. Our experiments demonstrate that BinArm is three orders of magnitude faster than existing fuzzy matching approaches.
- We propose the Weighted Normalized Tree Distance (WNTD) metric. This metric enables the comparison of execution paths between two given functions. It does so by comparing their respective acyclical CFG paths by taking into account a weight for each basic block in a given path. Weights are generated based on a basic blocks code contents and their relative position within the overall CFG.
- Rigorous experiments and benchmarks are performed on BinArm. These validate its performance both from efficiency and accuracy standpoints. These evaluations demonstrate that the framework can identify similar functions with a total accuracy of 0.92.
- BinArms capabilities are validated by performing meticulous experiments on real work IED firmware images. This process has identified 93 potential CVEs inside these images, the majority of which have been confirmed using manual analysis.

1.5 Thesis Organization

The structure for the rest of this thesis is as follows. Chapter 2 provides an overview of the smart grid and its specific areas that are of interest, namely distribution substations and the role IEDs play within them. Chapter 3 contains a thorough literature review and analyses on the state of the art relative to vulnerability identification in binaries. It also provides an overview of binary reverse engineering principles. In Chapter 4, we thoroughly discuss all aspects of the proposed methodology and its implementation. This chapter also covers all aspects related to the creation of our datasets which includes manufacturer, open-source usage, vulnerability and firmware identification. Chapter 5 then provides an in-depth evaluation that establishes a benchmark for the methodology along with its implementation. This is done through a set of rigorous experiments along with an overview of a possible real-world use case that aims at identifying vulnerabilities inside firmware from our dataset. Finally, Chapter 6 serves as a conclusion to the thesis, contains a summary of our contributions, an overview of identified limitations and provides ideas for future work.

Chapter 2

Background

A power grid is a complex and critical system whose purpose is to provide generated power to a diverse set of end users for consumption. A power grid system is generally composed of three main sectors: generation, transmission and distribution. All operations performed by each of these sectors take place in real-time. In the generation phase, electricity is created in a power plant. This power is then transferred to a nearby transmission substation that transforms the power to extremely high voltages. This is required to prepare it for long-distance transport; the higher the voltage, the smaller the amount of electricity lost over distance due to cable resistance. The electricity is then transmitted over high voltage power lines to a distribution substation. This substation performs the exact opposite operation than the other by transforming the received energy into a low voltage state. In this form, it can be transferred to local clients for consumption. The power can also go though a final stage of transformation before consumption in order to meet specific client requirements. Figure 1 illustrates this general power model.

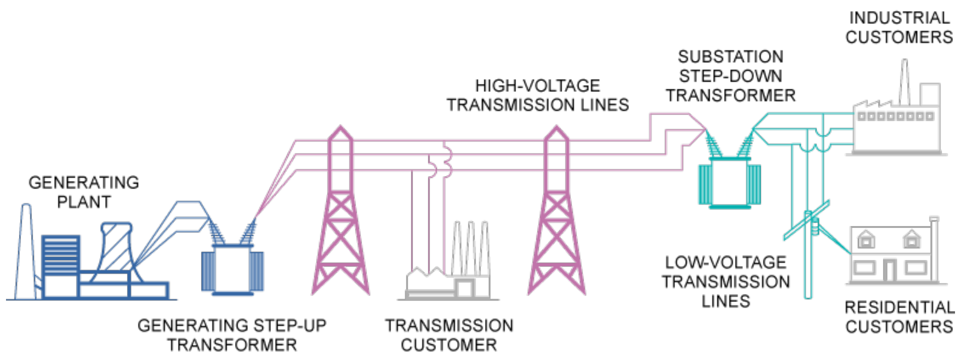

INDUSTRIAL CUSTOMERS
SUBSTATION STEP-DOWN TRANSFORMER
HIGH-VOLTAGE TRANSMISSION LINES
GENERATING PLANT
LOW-VOLTAGE TRANSMISSION LINES
RESIDENTIAL CUSTOMERS
GENERATING STEP-UP TRANSFORMER
TRANSMISSION CUSTOMER

Figure 1: General Power Model [27]

2.1 Smart Grid

For the vast majority of traditional power grids, the primary concern is with the reliability of the overall system [28]. This includes for example the capability to be resilient to unexpected equipment failure. However thanks to new developments, such as an increase in power demand over time, the diversification of energy sources with the introduction of renewables and the need for automation along with real-time situational awareness [29] other factors are gaining importance. Due to this, it is necessary to design and implement the next generation power grid, dubbed the smart grid.

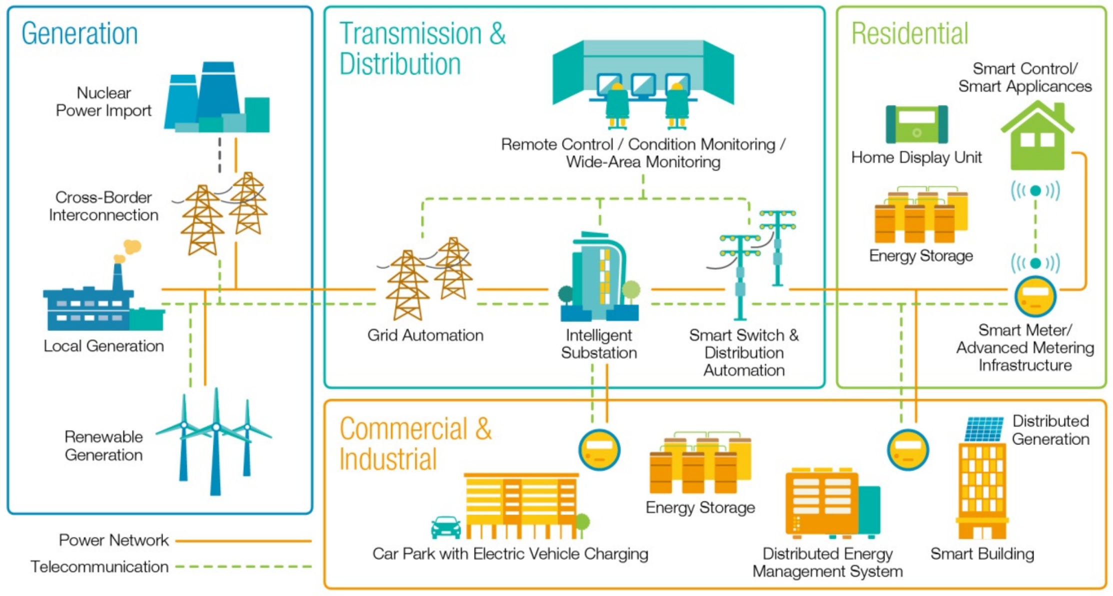

Generation
Nuclear Power Import
Cross-Border Interconnection
Local Generation
Renewable Generation
Transmission & Distribution
Remote Control / Condition Monitoring / Wide-Area Monitoring
Grid Automation
Intelligent Substation
Smart Switch & Distribution Automation
Residential
Smart Control/ Smart Applicances
Home Display Unit
Energy Storage
Smart Meter/ Advanced Metering Infrastructure
Commercial & Industrial
Energy Storage
Car Park with Electric Vehicle Charging
Distributed Energy Management System
Distributed Generation
Smart Building
Power Network
Telecommunication

Figure 2: Smart Grid Infrastructure [30]

One of the core concepts that the smart grid has introduced is an emphasis on communication and information exchange between deployed devices. Figure 2 illustrates this general idea. Thus, the smart grid introduces a layer of information and communication technologies above, and interconnected to traditional power grid networks. This results in an original remote measuring, monitoring, automation and control capabilities throughout the grid. Data acquisition enables the gathering and exchange of measurement or state information throughout the system. Monitoring leverages this newly acquired data to enable operators to have a far more complete idea of the current conditions and state that the overall system is in. Control involves the capability of sending remote commands that change the state of the system by operators or technicians. A real-world example of the benefits of such a change is in the time saved by being able to automatically or remotely open circuit breakers in a substation instead of having to send a technician over to do it manually. Another is the capability of utilizing collected data to perform state estimation in order to detect, in real-time, faults that occur throughout the entire power grid. By implementing such changes, it is possible to greatly increase the grids efficiency, reliability, and compatibility with renewable technologies.

The core of the smart grid related research in this work focusses on the distribution sector. Specifically, it considers distribution substations. Within these, it is desirable to identify the roles, responsibilities and security implications that IEDs have. These are all further explored in the rest of this chapter.

2.2 Distribution Substation

As aforementioned, the role of a distribution substation is to transform received high voltage electricity to a lower more suitable voltage for distribution to customers. Table 1, which is inspired by [31], provides a general overview of the primary components that are present in a traditional substation.

The introduction of the International Electro-technical Commission (IEC) 61850 [32] standard for substation automation has led to drastic changes in the substation landscape. Its intention is to leverage technologies such as Ethernet, high speed Wide Area Networks (WAN), and powerful but cheap computers to define a modern architecture for communication within a substation [33]. More importantly, this standard assures interoperability between all devices, irrespective of the manufacturer, in the substation automation context. The standard is meant to replace the older IEC 60870-4-104 standard and the use of DNP3 [34]. Such changes enable the creation of a Substation Automation System (SAS), a modern way to automate and control substation behavior. With this comes the introduction of a vast array of computers that provide the core of the new functionality. These are generally labeled as Intelligent Electronic Devices (IEDs). They are coupled with traditional ICS and power equipment to enable their integration into the network. The standard divides the digital topology of the substation into three different levels (process, bay and supervisory) that are interconnected via buses and communicate using three main protocols SV, MMS, and GOOSE [32]. Figure 3 gives a general overview of this structure and is inspired from [35].

Table 1: Main Traditional Substation Components

Component	Role
Capacitor Bank	• Helps maintain or increase the performance of electrical power systems by storing or releasing power • Switches between on or off depending on system requirements • Assists in keeping transmission system voltages stable during disruptions by providing voltage support
Circuit Breaker	• Used to connect and disconnect equipment such as transformers from buses • Triggered manually or remotely by an operator, or automatically in the case of a short circuit • Protects from current overload caused by a short circuit
Disconnect Switch	• Used to disconnect unused equipment from the rest of the network • Used to remove or transfer load • Used when replacing components or performing maintenance on them
Electrical Buses	• Used to link related equipment together • Joins different circuits together
Regulator	• Ensure that the quantity of received voltage remains constant for all powered devices • Minimizes equipment damage that can be caused by voltages that are out of acceptable range
Transformer	• Used to convert a certain power voltage into another and arguably the most important component of a substation • Come in two general varieties: Current Transformers (CTs) and Voltage Transformers (VTs) • Used in transmission substations (step up), and distribution substation (step down)

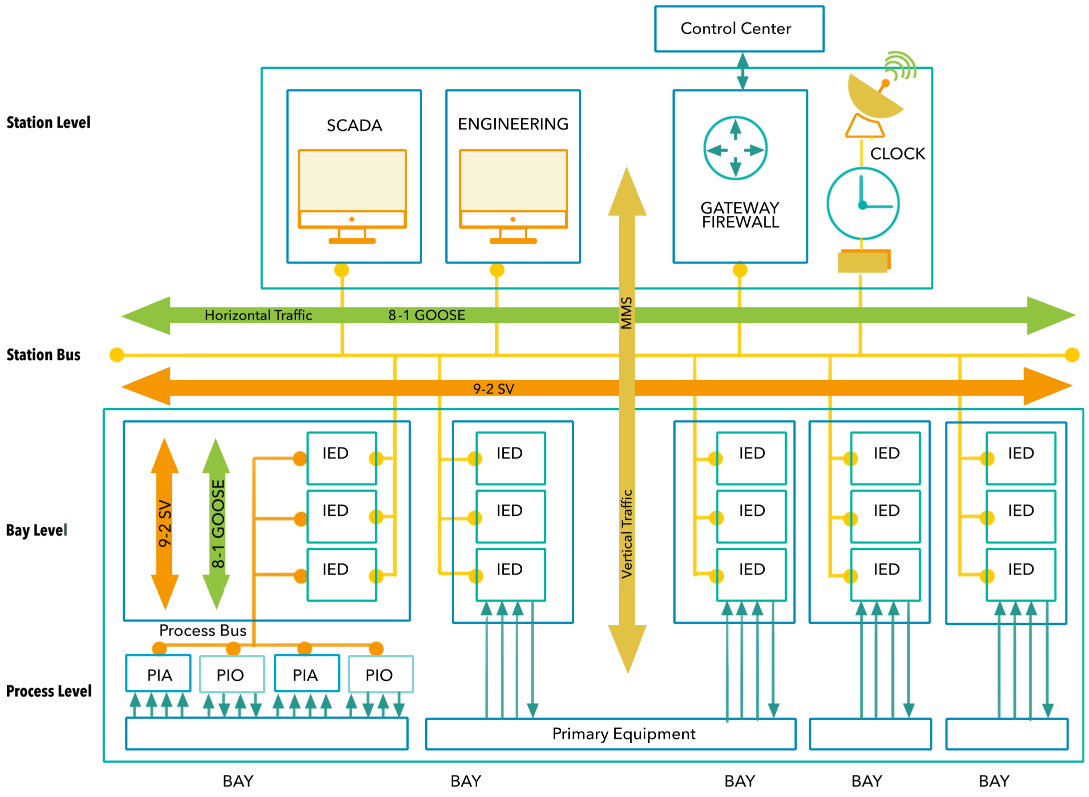


Figure 3: IEC 61850 Levels and Protocols

2.2.1 Substation Levels

Process Level: It is composed of all the traditional substation components. These expose binary and analog input/output sources that can be coupled with intelligent devices, sensors and actuators. Thus, certain traditional devices at this level are coupled with specific IEDs that are chiefly responsible for taking and transmitting measurements, or for receiving and executing remote commands.

Bay Level: It is composed of several physical devices, that contain IEDs, called electrical bays. These bays can be seen as the building blocks of the substation and must be adequately protected. In each bay specific control, protection and monitoring IEDs can be found. These are in turn connected to IEDs that are in the process level.

Station Level: It is where all centralized management takes place. This includes Supervisory Control and Data Acquisition (SCADA) servers. This is where all substation processes and state information can be visualized, analyzed and where commands may be sent from. This information is monitored by human operators or by automated systems. Operators can also interact with station level devices through Human Machine Interfaces (HMIs) that can store, handle and visualize amongst other things alarms, alerts and events.

2.2.2 Substation Buses

There are two different buses that enable the interconnection of all three different station levels.

Process Bus: The process bus enables communication between the process and bay levels. Thus, it is a network that connects the IEDs that are attached to primary equipment at the process level to higher level IEDs in the bay level. This bus is limited to the scope of a single bay and thus there is most likely several of them in any given substation. This communication system must provide constant hard real-time quality of service. An example of this network usage is the exchange of information between a breaker IED and a power and control IED.

Station Bus: It enables communication between the bay and station levels. Thus, the station bus is a network that connects the IEDs in the bay level to those in the station level. It also enables the communication between different bays on the same network. Such a configuration can connect up to hundreds of IEDs together. However, this network can be segmented into different zones. This communication system can provide soft real-time quality of service. An example of this networks usage is the exchange of information between a power and control IED and an HMI.

2.2.3 Substation Protocols

IEC 61850 introduces three main standardized protocols that allow the interoperability of different IEDs from different manufacturers within the same substation. These main protocols are used to tie all substation levels together. Each of them flows over certain buses and are used to transmit information such as remote control commands. Figure 4, which is inspired by [35], gives a general overview of the main protocol stacks used in IEC 61850 substation communications.

Sample Values (SV): The SV protocol is normally used on the process bus. It is used to exchange information between the IEDs that are connected to the primary equipment and bay level IEDs. This information is generally analog values, such as primary currents and voltages, that have

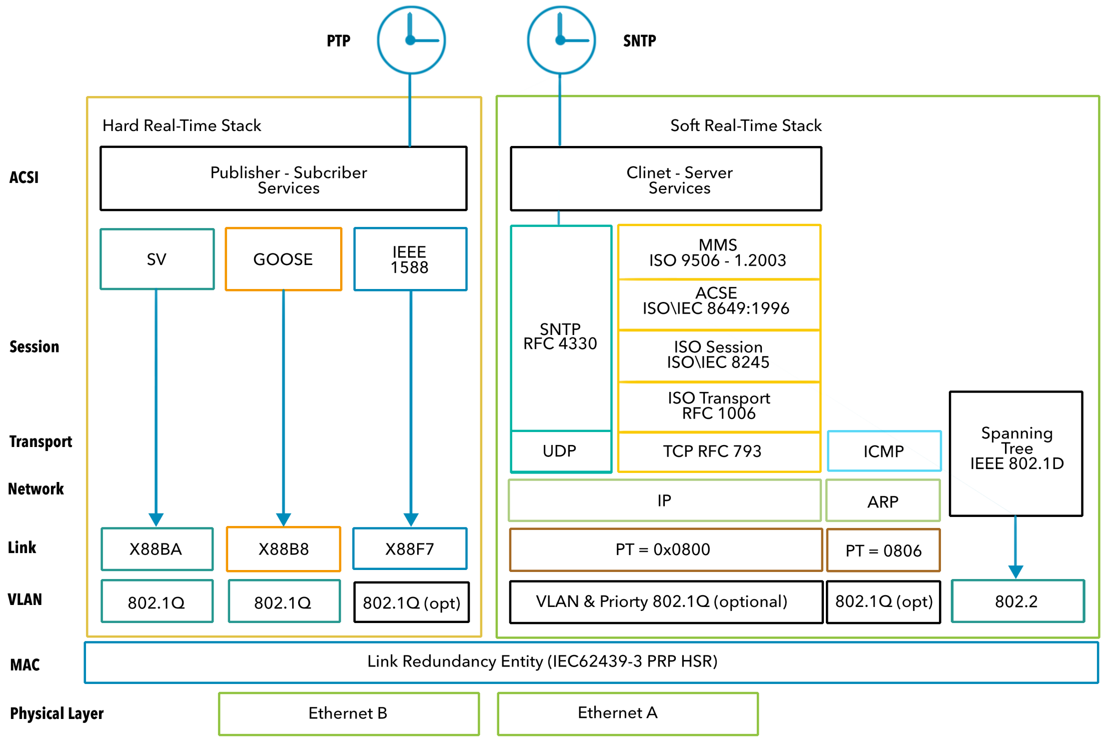


Figure 4: IEC61850 Protocol Stacks

been measured by sensors. Both layer 2 unicast and multicast are supported; however, unicast is rarely used. SV does not implement a retransmission protocol.

Generic Object Oriented Substation Event (GOOSE): The GOOSE protocol is used on both the process and the station bus. On the process bus, it enables inner bay IED communication. On the station bus it enables bay to bay level IED communications. GOOSE messages exchange critical system event information such as control signals and warnings [36]. This is enabled through the transmission of IEC 61850 Abstract Communication Service Interface (ACSI) objects (described in Section 2.2.4). GOOSE is an event based protocol that utilizes layer 2 multicast to transmit messages. Thus, when exchanging information there are two main roles a party can play, namely publisher or subscriber. When a certain event needs to be transferred by a given publisher, it will broadcast it on the desired bus. All parties on the local network will thus receive the message, but only interested subscribers will not discard it. GOOSE implements an extremely basic retransmission protocol that simply transmits the message at three time intervals with the hope that all concerned parties receive it at least once.

Manufacturing Messaging Specification (MMS): The MMS protocol is used on the station bus. It is used to enable bay level IED to station level device communication. The transmitted information is generally substation status information that is used for monitoring purposes [36]. MMS is a unicast client server based protocol that is transmitted over layer 3 TCP/IP. MMS servers generally reside on bay level IED devices. They receive requests from MMS clients, which are on station level devices such as an HMI. Upon receiving a request, they immediately respond with the requested information. MMS servers can also be configured to send push notifications to clients when specific events occur.

2.2.4 Intelligent Electronic Devices

IEC 61850 uses an object-oriented modeling style, dubbed Abstract Communication Service Interface (ACSI) to define different domain specific applications. It is a nested structure of four generic object types that represent the information exchange, type, functionality and data contained in a given device. As such, this model can be used to define all different types of IEDs. Figure 5, which is inspired by [32], gives a general overview of this structure.

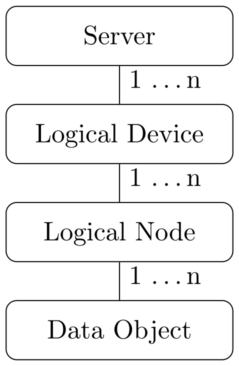


Figure 5: IEC 61850 Class Model

Data objects are made to hold domain specific information. They are composed of Data Attributes (DA) that are specifically defined data types. Several data objects can be contained inside a single logical node. Logical Nodes (LN) are a core object in this model. They represent the lowest level function that can be independently implemented in a given device. As such, each LN represents a standalone functionality that can be placed inside an IED. A list of all LN classes can be viewed online [37]. A Logical Device (LD) is composed of a set of LNs and is directly placed inside an IED device. It encapsulates all information and functionality that is required to perform a higher level capability such as the remote control of a circuit breaker. Finally, the server allows the external world to interact with the ACSI defined device. It contains and exposes the created model. As such, it is responsible of handling all incoming client requests and responding appropriately. It can return or modify specific attributes or generate reports with certain sets of data.

An example implementation of this scheme can be seen in Figure 6, which is inspired by [38]. In this example, an IED is composed to two LDs: Tampa_Protection and Tampa_Control. These are themselves composed to several data objects and data attributes [38]. Here a client could make a request for the Tampa_Control/Q0_XCBR.pos.stVal attribute to see if the circuit breaker (e.g., XCBR) that is connected to the IED is open or closed (e.g., pos.stVal).

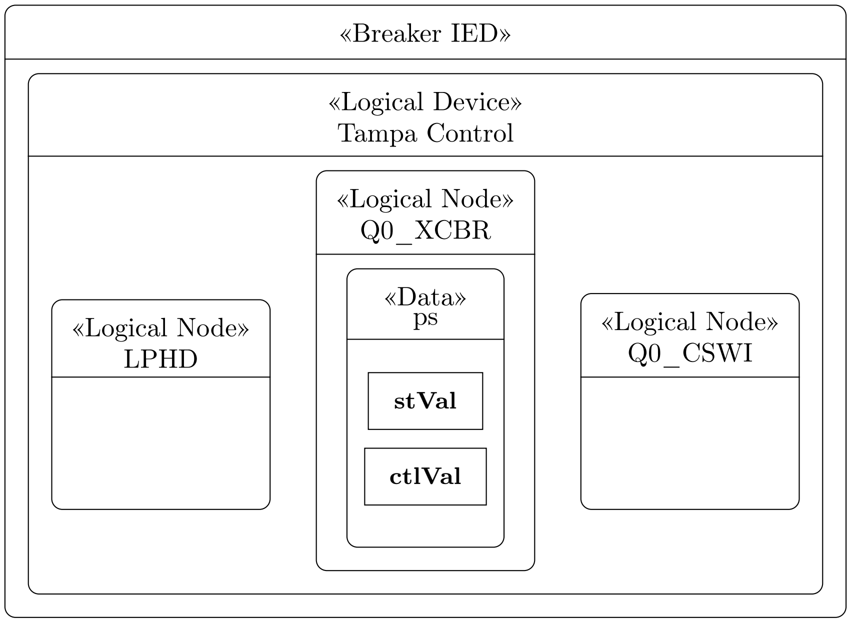


Figure 6: IEC 61850 Class Model Example

Due to this scheme, a very large range of different IED devices can theoretically be created. It is up to manufacturers to combine all of these concepts together as necessary to create an interesting product. However, despite this, there has emerged three general non exclusive categories that an IED can belong to. Table 2 gives an overview of these and their respective characteristics. To contrast this, Table 3 provides a list of some types of IEDs along with their respective characteristics. IEDs are generally implemented as embedded devices that leverage varying degrees of firmware in order to operate. This is further discussed in Section 2.5.

Table 2: IED Types

Type	Station Level	Protocol	Role
Control	Bay	GOOSE	• Used to remotely send and receive commands that control system behavior, such as opening a given circuit breaker due to a detected fault • Commands are generally received through the station bus from the command center and forwarded to primary devices through the process bus
Monitoring and Relay	Process, Bay	GOOSE, SV	• Responsible to convert received analog input (e.g., currents, voltages, power values) from primary equipment into a digital format that can be used throughout the network • Digitized data is formatted and then sent to higher level devices in the bay, and can be used to derive statistics such as peak values
Protection	Bay	GOOSE, MMS	• Detection of faults that then need to be isolated from the network in a specific and timely manner • Impacts of faults include equipment damages, degradation of service and loss of production

Table 3: IED Examples

Name	Type	Role
Bay Control IED	Control	• Enables the management of one or more bays • Combines and can perform the tasks of several other specialized IEDs • Functions include managing switchgear, breakers, or transformers
Breaker IED	Control	• Receives and executes commands relative to breaker operation such as to open, or close • Indicates the state that the breaker is in • Sends and receives data through the process bus to power and control IEDs
Merging Unit (MU)	Monitoring and relay	• Used to collect analog measurement data from process level devices, and convert it to a digital signal • Transmitted to other IEDs at the bay level using SV or directly to the control center using GOOSE • Relevant process level devices include Con Conventional Instrument Transformers (NCITs) (e.g., non conventional CTs or VTs)
Recloser Protection IED	Protection	• Used to supervise the operation of overhead lines and power feeders in a power system • Increases reliability and maintainability of protected systems
Transformer Protection IED	Protection	• Used to supervise the operation of power transformers • Provides protection from transformer faults • Helps extend the lifetime of the associated transformer

2.3 Smart Grid Security

The advent of the smart grid is estimated to introduce millions of new intelligent and interconnected devices into classical power systems [39]. These immense changes to the power grid infrastructure insert several new attack surfaces that have not been previously present or accessible. Some of these new risks can be seen in Table 4, which is inspired by [39]. As such, the importance of cyber security and cyber-physical security can no longer be overlooked. Thus, it is no longer an option for operators and manufacturers to take a *security by obscurity* approach when it comes to their systems or communication protocols [28]. In fact, these weaknesses are a threat to the overall reliability and stability of power systems. If left undefended, they become extremely vulnerable to sabotage, cyber-espionage, or even cyber-warfare attacks. These real-world threats can be seen through attacks performed by hackers [40] or specialized malware such as previously mentioned Stuxnet [3].

Table 4: New Smart Grid Risks

Risk	Description
Complexity	• An increase in complexity leads to the potential for the introduction of more vulnerabilities and oversights that malicious entities can exploit • This is at all levels such as deployment scale, device communications, and software
Exposure	• The increase in interconnectedness of smart devices exposes them to new external threats • The use of an isolated network zone is no longer possible or acceptable
Information	• With an increase of information exchange, there is a risk of attackers targeting it • This can be through stealing or modifying sensitive information such as customer electricity usage data

Several organizations have taken initiative, amongst their other responsibilities, in assisting with the defense of smart power grids from malicious entities [41]. Some of these include the Electrical Power Research Institute (EPRI) [42], the International Electrotechnical Commission (IEC) and the North American Electric Reliability Corporation (NERC) [39]. These entities can issue security standards and recommendations that the general industry should or legally must comply to.

2.4 Substation Security

Information securities traditional triad of objectives are availability, integrity, and confidentiality. Depending on the context, these properties are extended to include many others, such as anonymity, and may vary in importance. Table 6, which is inspired by [34], shows which objectives are the most important in the substation context and contrasts them to those that are not. Out of all these objectives, communication network availability is the most important [39]. This is due to the fact that if a given substations communication network goes down, all functionalities provided by it are rendered inoperable. This includes the loss of all monitoring and control capabilities, and can lead to a stations complete failure.

Table 5: Prominent Substation Security Standards

Name	Description
IEC 62351: Security Standard for Substation Automation	• Designed to highlight cyber-security procedures that should be used in the smart grid • Focusses on communication protocols and how to secure them • Composed of eleven parts, each of which focusses on specific aspects of security features that should be implemented [43]
NERC: Critical Infrastructure Protection (CIP)	• Large-scale security standard that is composed of a set of standards, eleven of which are currently enforced [44] • Attempts to unilaterally standardize all aspects pertaining to power system security in North America • Focusses on critical cyber asset identification, risk analyses, physical security, and staff training [45]

Due to the highly essential nature of substations, it is necessary to address its security objectives, and many other concerns. This includes several facets such as network and device security. Prominent standardization bodies have thus created a large set of security recommendations in the attempt of preventing potential attacks. Table 5 briefly discusses two of the most prominent North American ones. It is also important to take into account standards that have been made for security in general control systems environments. This is because their general architectures have similarities with those of the smart grid [43]. However, these essential properties cannot be implemented to the detriment of the overall systems functionality or efficiency. An example of this is in time-critical systems, where the process of encryption is too costly. Instead, they only require data integrity and authenticity [46].

Table 6: Substation Security Objectives

Name	Reason
Availability	• Power systems must always be running • Any failure, natural or malicious, can lead to severe consequences such as power outages or damaged equipment
Integrity	• It is critical that all data exchanged over a substations network be unmodified by malicious parties • If this is possible, attackers could easily manipulate the behavior of the overall system to make it operate as they desire • For instance, an attacker could modify readings to simulate a fault
Non-repudiation	• It is essential to be unable to deny the authenticity of messages that are sent over the substation network • An example of this is the need for non-repudiation in the transmission of MMS transactions over the network [46]

2.4.1 IEC 61850 Standard

Despite the clear need for security in the smart grid context, the IEC 61850 standard basically side steps any security regulations or recommendations. As such, they provide very few, generic, and basic security guidelines. This includes the fact that cyber security must be ensured; to limit substation network access to only authorized parties and to limit network access though the use of gateways. Other mentioned security oriented technologies include the use of syslog, security audit trails, passwords, access control, port security and encryption [32]. In fact, it does not specify any concrete security procedures but leaves these to other standards to handle. Specifically, this responsibility has fallen into the hands of the IEC 62351 [47] standard and is further discussed in Subsection 2.4.2. This lack of standardized security measures has created the opportunity for other organizations, and researchers to identify weaknesses in all aspects of the standard, and make them publicly known. These works target many aspects of an IEC 61850 substation such as its protocols, networks and components.

The National Electric Sector Cybersecurity Organization Resource (NESCOR) objective is to further the integration, and validation of security within the smart grid. They do so though partnerships with bodies such as the United States Department of Energy (DOE) and multinational experts [48]. One of their major contributions to smart grid cyber security is the NESCOR failure scenarios [49]. These explore a vast set of realistic attack scenarios that can take place on the smart grid. This includes all facets such as Distributed Energy Resources (DERs), Automated Metering Infrastructure (AMI) and substations. This expert generated knowledge can be used as a basis for a wide range of security oriented tasks. Examples of this are researchers who use this information to formulate possible attack scenarios in their work, or utilities to assist in tasks such as risk assessment [49]. Each scenario is classified and throughly discussed. This is done through a general description, a list of relevant vulnerabilities, the impact of the attack and possible mitigations. Critical scenarios often also contain fully-fledged high-level attack trees to assist with mitigation. Scenarios that are pertinent to substations can often be seen to be potentially taking advantage of weaknesses in the IEC 61850 standard. An example of such a scenario can be directly taken from the NESCOR Failure Scenario, and is denoted as DER.6. This scenario describes an attack that abuses remote commands (e.g., replay attack) in order to cause a DER system imbalance. This effectively results in tripping off all clients associated to the affected substation.

2.4.2 IEC 62351 Standard

The IEC 61850 standard does not specify any concrete security procedures. These are left to other standards to handle. Due to the increased use of the IEC 61850 standard, it is essential that such concerns be fully and properly addressed. This has led to the introduction of IEC 62351, a standard for security in the communication protocols that are defined by the IEC Technical Committee (TC). As such, this includes all those proposed by the IEC 61850 standard. The IEC 62351 standard is composed of fifteen parts [50], each of which focusses on specific aspects of security features that should be implemented or tested. Table 7, which is inspired by [43, 47], briefly describes the purpose of each of these sections.

Table 7: IEC 62351 Parts

Part	Description
1	• Provides an overview of the entire standard along with attacks and defenses
2	• Defines all the domain specific vocabulary that is used throughout the standard
3	• Focusses on the use of IEC 61850 TCP/IP based protocols, such as MMS, over TLS • Enables them to acquire integrity, authentication and confidentiality properties
4	• Covers general security practices that should be used along with IEC 61850 MMS • Introduces the use of digital signatures, and mutual authentication, along with general integrity, and confidentiality capabilities
5	• Focusses on further securing IEC 60870-5 based transmission protocols, such as Distributed Network Protocol (DNP3) • Enables security properties such as authentication, along with protection against spoofing, modification, and replay attacks

Part	Description
6	• Relates to improving security in the context of the IEC 61850 GOOSE and SV protocols • Focusses on providing authentication, and integrity (e.g., RSA signatures) for all exchanged messages
7	• Provides a set of Network and System Management (NSM) objects that represent the state of an IEC 61850 substation • Used to detect and mitigate attacks such as malicious packet injection
8	• Focusses on entity, role and authentication management (e.g., RBAC) • Recommends several core RBAC principles such as least privilege
9	• Provides specifications for cryptographic key management procedures • Covers both symmetric and asymmetric keys, along with key generation, distribution, renewal, and revocation
10	• Provides power system security architectures that cover technical aspects, recommendations, operational general guidelines, and responsible personnel • Presented in the form of real-world deployment examples

Part	Description
11	• Focusses on different ways of securing XML documents that are used and exchanged throughout all IEC standards scope • Leverages pre-existing W3C XML security standards

As can be clearly seen, the implementation of the IEC 62351 standard within a IEC 61850 substation can greatly increase its security standing. This can be observed by comparing the effectiveness of the previously described attacks on IEC 61850 protocols before and after the security measures have been applied. The delay attacks described in Lu *et al.* [51] can potentially be detected and mitigated with the use of the prescribed NSM objects. Through the use of the TLS protocol, attacks like ARP cache poisoning proposed by Kush *et al.* [52] can be prevented. Furthermore, the spoofing and injection attacks proposed by Hoyos *et al.* [53], Valdes *et al.* [54], and Hong *et al.* [55] can be mitigated using the suggested authentication, and encryption schemes. However, these improvements though essential, do not cover all known system vulnerabilities. For example, this standard does not ensure that all NESCOR failure scenarios are protected against. An example of this is when considering the previously discussed DER.6 scenario in Subsection 2.4.1. Assuming that the GOOSE protocol is utilized to control a given DER system, it is possible depending the implementation, that there is integrity but not confidentiality protection. This is because confidentiality is not required due to time constraints. As such, there is the hypothetical possibility of an attack. For instance, since the attacker knowns the contents of each packet, he can drop or manipulate specific ones with the intent of causing DER imbalance [53].

2.4.3 Intelligent Electronic Device Security

With the introduction of a wide range of embedded devices into the substation environment, comes an increase in attack surfaces and vectors. It is often the case the deployed IEDs are not updated and utilize code that contains known vulnerabilities [56]. These devices are networked in order to be remotely controlled. This opens up a wide range of attacks such as spoofing, command injection and malicious firmware updates [57]. They are susceptible to malware [58], some of which can even be pre-installed on deployed devices [59]. Furthermore, they sometimes do not have any authentication [60], which allows malicious entities or employees to manipulate the devices [56]. Clearly, it is often the case that the security concerns of IEDs are barely taken into account [61].

In light of this information and the critical nature of the smart grid, it is essential to mitigate the aforementioned security vulnerabilities. These solutions must place an emphasis on ensuring proper authentication, authorization and integrity schemes [56]. One proposed approach is to design and implement secure deployment architectures that are tamper resistant, remotely recoverable and securely updatable [57]. LeMay *et al.* [62] propose an example of one that can be applied to the IED context. Dubbed cumulative attestation kernel, it aims at providing firmware auditing structure that is implemented in a remote and secure fashion [57]. It is however extremely difficult to develop blanket solutions that can ensure IED and other embedded device security [63]. This is primarily due to the complexity and diversity of deployments. Contributing factors include device diversity (e.g., vendors, functionality, implementations), implementation constraints due to requirements [64] and scalability [63]. Due to this, there is a wide range of specialized analysis techniques that can be used to better IED device security. It is possible to analyze a device's firmware in order to identify weaknesses it contains. A recent example of such an attempt is performed by Kwon *et al.* [13] where they search for known vulnerabilities in smart grid devices. Such an approach is however not prevalent due to its high degree of difficulty [65]. Pour *et al.* [66] suggest the use of an IED specific rule based IDS that is based off the IEC 61850 standard. Another possibility is to diversify the range of deployed device manufacturers and firmware in order to complicate large-scale attacks [56]. This solution is unfortunately often too expensive and time consuming to implement.

Another critical aspect is to secure all communication channels. In order to exchange information, IEDs leverage the previously discussed substation communication protocols. Several research proposals have examined the weaknesses is such communication mechanisms. Kush *et al.* [52] demonstrate an IED GOOSE based DoS attack. They describe their proposed attack as GOOSE poisoning. This attack utilizes malicious GOOSE packets whose status numbers are manipulated in order to make them higher than current non malicious packets. This tricks the IED into dropping all packets (e.g., valid ones) whose sequence numbers are lower than the malicious packets. This basic principle is used to implement two other attack variations namely a high rate flooding attack and a semantic attack. Valdes *et al.* [54] focus on the abuse of protective relay IEDs. Under normal circumstances, these relays are configured to detect faults and react to them through tripping the appropriate breakers. The authors intent is to create malicious GOOSE or SV packets. This is either done from the ground up or by modifying intercepted ones. These packets can then be injected back into the network, and used to trick the system into believing there is a fault. Such an attack will result in the triggering of protective relays, which will potentially lead to power outages. If such an attack is carefully designed to concurrently target multiple critical protective relay IEDs, it could result in large-scale power outages. Very similarly, Hong *et al.* [55] take advantage of IEDs and circuit breakers by sending them crafted commands. This achieved by taking advantage once again of the fact that SV and GOOSE protocols do not have any integrity or replay prevention capabilities. Thus, they are able to perform replay attacks that maliciously open circuit breakers. If well coordinated, a large-scale attack that leverages this approach could lead to massive power outages.

These concerns have been taken into account by standardization bodies who have produced more specialized IED cyber security standards. These are further discussed and presented in Table 8. The most relevant of these within the current context is IEEE 1686-2013 [67]. It describes IED specific cyber security requirements and is divided into seven sections. These sections are each further discussed in Table 9.

Table 8: Relevant Smart Grid IED Security Standards

Name	Description
IEEE 1402-2000: Guide for Electric Power Substation Physical and Electronic Security [68]	• Highlights that all physical devices in substations, including IEDs, are vulnerable to physical and cyber intrusions • Proposes different security procedures to assist in mitigating these • Evaluates the effectiveness of the proposed mitigations
IEEE 1686-2013: Standard for Intelligent Electronic Devices Cyber Security Capabilities [67]	• Provides all implementation details necessary to conform to NERC CIP IED security requirements [44] • Designed to be generic and applicable to any IED • Covers topics such as IED access, operation, configuration, firmware revision and communication

Table 9: IEEE 1686-2013 Parts

Part	Description
Electronic Access Control	• Defines mandatory and uncircumventable device user identification and password requirements • Mandates the use of RBAC to manage individual user capabilities • Defines a set of sensitive operations that require specific RBAC roles to perform
Audit Trail	• Mandates that certain system events must be recorded • Provides event storage format and integrity requirements • Defines a set of system events that must be logged
Supervisory Monitoring and Control	• Mandates that IEDs report events (authorized activities) and alarms (unauthorized activities) to its supervising system • Defines a set of events and alarms, along with how to group them • Defines how a supervising system can control IEDs along with its required permissions to do so
IED Cyber Security Features	• Defines which protocols should be used to enable IED network communications • Enforces the use of current NIST cryptographic requirements • Enforces the use of encryption as defined in IEEE 1711

Part	Description
IED Configuration Software	• Enforces the use and validation of digital signatures for all configuration files and firmware updates • Defines two configuration viewing and modification modes that require the proper privileges to access • Defines four roles that have varying configuration modification privileges
Communications Port Access	• States that all ports can be enabled or disabled via configuration • Stipulates that disabled ports must ignore all communications • Stipulates that all unused ports must be disabled
Firmware Quality Assurance	• Enforces the use of firmware quality assurance as defined in IEEE $C37.231$

2.5 Firmware Reverse Engineering

Previously discussed smart grid components, such as IEDs and routers all rely on firmware in order to operate. However, firmware is not limited to these contexts and can be found in a large range of devices such as smart home appliances, vehicles and networking devices. It is furthermore possible to analyze the contents and the behavior of a given firmware image using reverse engineering techniques and principles. In the context of this work, it is necessary to have a general understanding of both the underlying principles of firmware along with their disassembly and analysis. This is due to the fact that such steps play a crucial role in the identification of vulnerabilities contained within firmware.

2.5.1 Firmware

Traditionally, firmware has been viewed as software in the form of microcode that is stored in a read-only format. This links the device hardware to the CPU instruction set [69]. It is also traditionally executed in very computationally limited environments. However, with the evolution of computational devices, this definition has further expanded. It is now more generically defined as the aggregation of data and software. It is stored in a read-only format that interacts with underlying hardware [70]. Thus, firmware differs from traditional software in that it is highly coupled to the underlying hardware implementation and is used to provide an optimized and consistent interface to interact with it [71].

Due to their high-level of coupling with CPU instruction sets, firmware is specifically tied to the CPU architecture that it is designed or compiled for. Some of the most popular CPU architectures for embedded devices are ARM, X86, and MIPS [72]. Any given firmware image can be generally composed of five main components. These are further discussed in Table 10, which is inspired by [73, 74]. Depending on the complexity and the purpose of a given firmware image, each of these components may or may not be present. Based on this criteria, it is possible to extrapolate three different firmware image classification categories [73]. A full firmware image contains all main components and is the most complex form of firmware. A similar but more simplistic implementation is the integrated firmware. It contains a primary application that runs with a bare-bones kernel. These two types of firmware can be considered as stand-alone and can be directly flashed onto an embedded device. The third type is a partial firmware image that only contains a small set of resources. This is meant as an update that only replaces part of the data that is already present in a given device.

2.5.2 Reverse Engineering

Reverse engineering is the process of disassembling an object in order to determine its structure or behavior. This knowledge can then be further leveraged, amongst other things, in order to further analyze, attack, reproduce, or improve the object in question. As such, these deconstruction procedures can be used for both malicious and benign intents. In the context of firmware, such procedures are often used to perform defect identification, tempering, and malicious attack development [75]. The first phase of this process is generally to acquire a firmware image. Once this has been completed, there are several steps and procedures that can be followed in order to successfully analyze it. Each of these steps involves the use of a diverse set of techniques and tools. Generally speaking, firmware reverse engineering is a lengthy and complicated task due to the large amount of required domain knowledge [15]. All tools referenced in this section, along with several other prominent ones, are further described in Table 11.

Table 10: Main Firmware Components

Component	Example	Description
Binaries	ELF	• Executable files, libraries, and applications that are stored in the file system
Boot Loader	U-Boot	• First piece of software to be run upon starting the device • It is responsible to load the kernel into memory and to ensure it can be properly run
File System Image	YAFFS, JFFS2	• Contains the storage structure along with all files that are kept in the embedded device • Extractable using specialized tools to recover the information they contain
Kernel Image	Linux, Windows CE	• Is the actual operating system that is run on the device. It is stored in some form of non volatile memory • It is the first layer of abstraction and is responsible the foundation for system • It provides several capabilities such as memory management and networking
Resource Files	DOC, PNG	• General files that are stored in the file system • They provide configuration, information and support for device functionality

2.5.2.1 Acquisition

The difficulty of acquiring a given firmware image can greatly vary. On one end, it is possible to extract a vast amount of firmware from the public internet. This can be directly from a manufacture's website, or a third party source such as a FTP search engine. To greatly accelerate the process and quickly gather large amounts of firmware, a web scraper or crawler can be created. Another simple way is to contact the manufacturer, and request a copy of a given firmware for security evaluation or educational purposes. They may simply provide it, especially if you have purchased a given embedded device. However, such simple approaches are not always available. It may be necessary to directly extract or dump it from a device's chip memory. This can be achieved through several different ways such as an EEPROM programmer, bus monitoring during code upload and schematic extraction [76]. Difficulties in performing this task include hardware locks and component interference. This can be resolved by physically modifying the original hardware, or manipulating the boards circuits using probes in order to make it behave in a desired way [76].

2.5.2.2 Unpacking and Extraction

Once a firmware image has been acquired, its analysis can take place. Reverse engineering of firmware can be seen as an iterative process whose purpose is to completely acquire all encapsulated raw information. Depending on the format of the acquired firmware archive, it first has to be unpacked. Assuming an archive file has been retrieved from a manufacturer's website, this process can be achieved using tools like 7z [77], and xz [78]. The result of this process is generally a set of miscellaneous files such as a flashing tool, along with the actual firmware image. From there, the extraction process can repeat for the image. This generally results in a subset of the main firmware components discussed in Table 10. Furthermore, the file system can be extracted using tools such as unyaffs [79]. It is generally of particular interest to acquire all data that the identified file systems contains, such as binaries, for further analysis.

This process although seemingly simple at a high level, has several complications. It is often the case that the format of the received data is unknown and cannot be easily or directly used by most tools. As such, it is necessary to utilize means such as binwalk [80] to identify header signatures, along with interesting data segments in data files. If these tools do not succeed, it is necessary to manually inspect the files hex code, using tools such as bless [81], and dd [82] to extract relevant data chunks. Performing such tasks is extremely difficult due to the large requirement of time, domain specific knowledge and research [15]. Furthermore, binaries are often obfuscated or encrypted for protection. These processes effectively make it extremely difficult (e.g., obfuscation), or even impossible (e.g., uncrackable encryption) to directly access the contents of a given blob. Encrypted binaries can sometimes be identified by their use of specific headers. An example of this are files encrypted with openssl that start with the first 8-byte signature "Salted__". Tools that can perform this encryption signature identification process automatically include signsrch [83]. Another technique is to use tools such as strings [84] to see if there are any human readable strings in the given data. If there are, the data is not encrypted.

2.5.2.3 Firmware and Binary Identification

Once all information has been unpacked from the firmware image, it needs to be filtered through for all relevant information. Depending on the context this can include binaries, configuration files, embedded files, and of course, the firmware. In this phase, it is necessary to leverage tools that perform file signature matching. This greatly increases the overall time required to perform this task. Relevant tools include binwalk, signsrch and file [85].

2.5.2.4 Disassembly

Disassembly is the process of converting a given binaries machine code instructions to their higher level symbolic representation [15]. This is applied here by converting a given firmware binary from its machine code format into a more understandable assembly (ASM) one. This process is performed by utilizing powerful frameworks such as IDA Pro [86]. It is a highly versatile multi-processor disassembler and debugger. As such, it is generally able to automatically disassemble most binaries with close to no effort on behalf of the user. Once all code blocks have been identified, IDA Pro enables further analysis through different graph views, and its powerful Python interface [87]. Unfortunately, it is often the case that disassemblers are not able to automatically disassemble a given firmware or binary. This can be caused by several reasons, some of which are further discussed.

Obfuscation: This is the transformation of a programs semantics that makes it harder to reverse engineer, while maintaining its functionality. This can be achieved using techniques such as indirect memory pointers, padding byte injection and indirect jumps [15]. This is particularly troublesome for such techniques often trick dissemblers into producing the wrong output. For example, Linn *et al.* [88] demonstrate that this is possible by injecting junk bytes into binaries that do not affect their functionality or execution, but cause disassembler errors. Possible solutions include attempting to reconstruct the programs control flow and analyze it using statistical methods to ensure its validity [15]. Overall, getting past such defenses is a very complicated process that often results in the abandon of the reverse engineering effort [89].

Hardware Architecture Identification: Determining the CPU architecture for a given firmware is a necessary step in its analysis process. This allows you to determine the instruction set mappings that are to be utilized in order to further disassemble it. This process is generally done by searching the binaries byte sequences in order to identify operation code signatures. These signatures specifically pertain to given CPU architectures and instruction sets. If frameworks such as IDA Pro are not sufficient to do so there exist other powerful alternatives such as binwalk. It is also possible that these tools do not natively contain required identification signatures. Thus, it may be necessary to create new context specific ones [90]. Such a process requires a very high-level of domain knowledge and is not easily performed.

Table 11: Prominent Reverse Engineering Tools

Category	Tools
Archive Extraction	binwalk [80], lzma [91], tar [92], unrar [93], xz [78], 7z [77]
Binary Utilities	BAT [94], binutils [95], cpu_rec [96], dd [82], objdump [97]
Disassemblers	disassembler.io [98], ERESI [99], Hiew [100], Hopper V4 [101], IDA Pro [86], Immunity Debugger [102], PE Explorer [103], Radare [104], Relyze [105], x64dbg [106]
File System Extraction	binwalk [80], cnu_fpu [107], squashfs-tools [108], ubifs [109], unyaffs [79]
Hex Editors	Bless [81], hexdump [110], WinHex [111], wxHexEditor [112]
Signature Identification	binwalk [80], file [85], hash-identifier [113], PEiD [114], signsrch [115], strings [84], trid [116]

Entry Point Discovery: An entry point is the location that marks the transfer of execution control from the operating system over to another program. In this scenario, this is the specific location in a given firmware binary that is first fed to the processor in order to start its execution. In other words, it is the base address where the binary should be loaded. A given binary blob can contain several entry points [17], and it may not always be possible for frameworks such as IDA Pro to automatically identify them. This inability to automatically identify them through traditional means can be due to their lack of a standard file format [16]. Shoshitaishvili *et al.* [17] propose a possible way to bypass this problem. They first scan the given binary for function prologue instructions that correspond to a desired CPU architecture while searching for a terminating return instructions. This marks the presence of a given function. When all functions are identified, they are placed inside call graphs. The roots of these graphs are then treated as possible binary entry points. Zhu *et al.* [11] propose another way of circumventing this problem. It is specifically related to the ARM architecture. First, they specifically focus on the identification of string and their offsets within a given firmware image. They then filter their results for all those that are loaded specifically using the LDR mnemonic and identify the addresses that are used to reference them. Next, they leverage the identified string addresses and offsets to calculate the image base. This is because the address of a string loaded in memory is the image base plus its offset.

Chapter 3

Related Work

One of the core aspects of this work focusses on the identification of vulnerabilities inside given IED firmware binaries. Although novel in this regard, general binary analysis procedures, methodologies and frameworks have been previously created. It is thus necessary to acquire an overview of the most prominent endeavors in this field.

3.1 Prominent Binary Analysis Taxonomies

We propose two large taxonomy groups that aim at classifying both the purposes and methodologies of all examined work. This provides a high-level overview and comparison of the current state of the art.

3.1.1 Purposes

It is interesting to create a taxonomy of all identified works based on their purposes. This provides a high-level overview of all prominent and currently available binary analysis approach capabilities. Figure 7 gives an overview of the different purposes each identified technique provides. All identified taxa are then further defined.

3.1.1.1 Firmware Analysis

Broadly, all relevant and prominent algorithms can be grouped into two main categories. The first of these is firmware analysis. It represents all techniques that focus on the evaluation of firmware samples. Specifically identified approaches are further discussed.

Binary Search Engine: This involves the identification of a specific code segment, for example a function, within a large pool of candidates. This process should generally return a set of possible matches, each of which has a confidence score. This methodology has several prominent applications that include (i) bug search, (ii) code-reuse and (iii) copyright enforcement. *Bug search* utilizes this process to specifically identify code segments that are known to contain vulnerabilities. *Code-Reuse* is more broadly defined as identifying any duplicate code segments of interest. *Copyright enforcement* is particularly concerned with the identification of duplicate code segments that infringe upon copyrights.

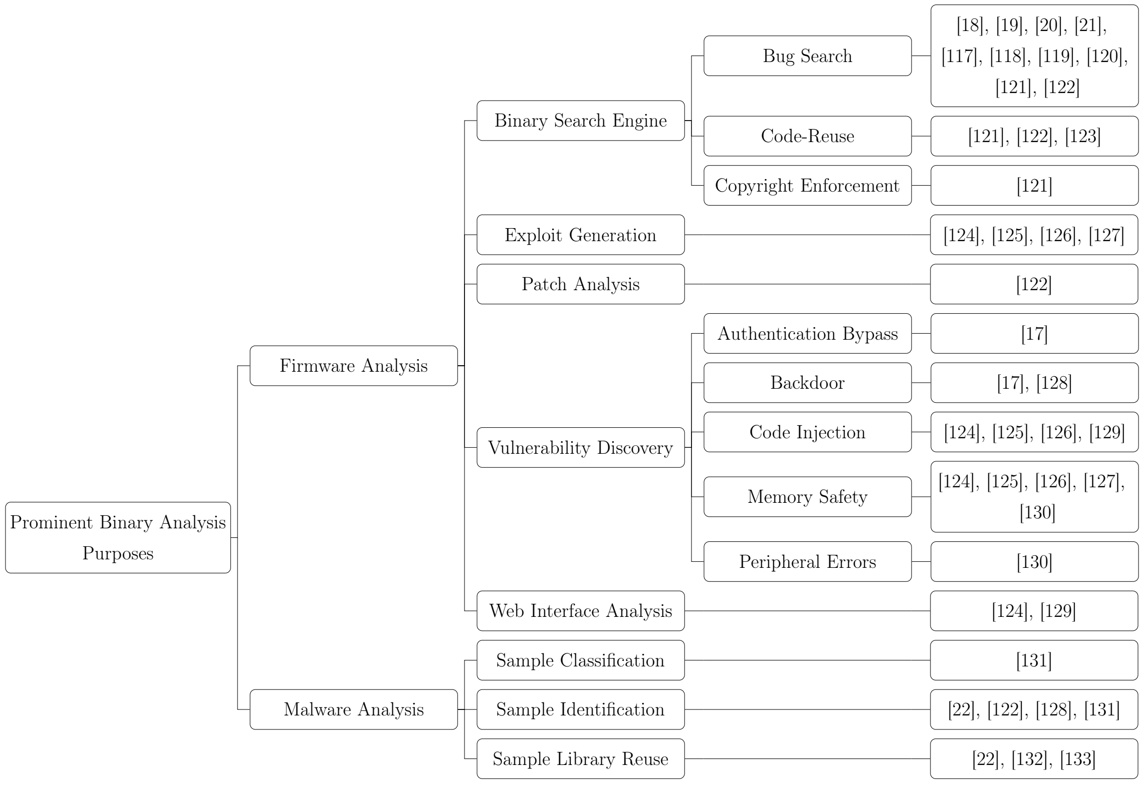


Figure 7: Prominent Binary Analysis Purposes

Exploit Generation: This analysis approach distinguishes itself by attempting to automatically generate exploits for identified vulnerabilities. This technique has the potential, if properly implemented, to relatively easily identify and exploit zero-day vulnerabilities.

Patch Analysis: This task involves the analysis and comparison of a given binary both before and after it has been patched. This can be used for a few reasons, such as validating that it has resolved the underlying issue or to reverse engineer the vulnerability if it is not publicly disclosed.

Vulnerability Discovery: It is the process of analyzing the behavior of binaries in order to identify vulnerabilities. This approach differs from bug search in that it does not require a large pool of candidates to perform comparison.

Web Interface Analysis: Embedded devices often contain web servers that provide relatively simple interfaces. These also need to be evaluated for potential security weaknesses or flaws.

3.1.1.2 Malware Analysis

The other category of interest is malware analysis. It can be defined as all approaches that focus on the analysis and identification of malware samples. This can be within any form of binary including firmware and shared libraries. Although firmware analysis is of more direct interest, some methodologies proposed by identified malware analysis research are of interest and remain applicable to the current context.

Malware Sample Classification: By analyzing given malware samples, it is possible to extract certain characteristics that enable their classification into specific groups. These characteristics are often behavioral-based.

Malware Sample Identification: The purpose of this process is to identify malware within a given target binary. Can be viewed similarly to the binary search engine scenario.

Malware Sample Library Reuse: One critical task in malware analysis is to identify its behavior. This process can be assisted by identifying which libraries and functions it leverages. These provide indications as to its capabilities.

3.1.2 Methodologies

Each identified work leverages a set of techniques when it comes to performing the purposes it boasts. These can be broadly divided into three main binary analysis categories; namely dynamic analysis, hybrid analysis and static analysis. Figure 8 gives an overview of all proposed methodologies. All mentioned taxa are then further defined.

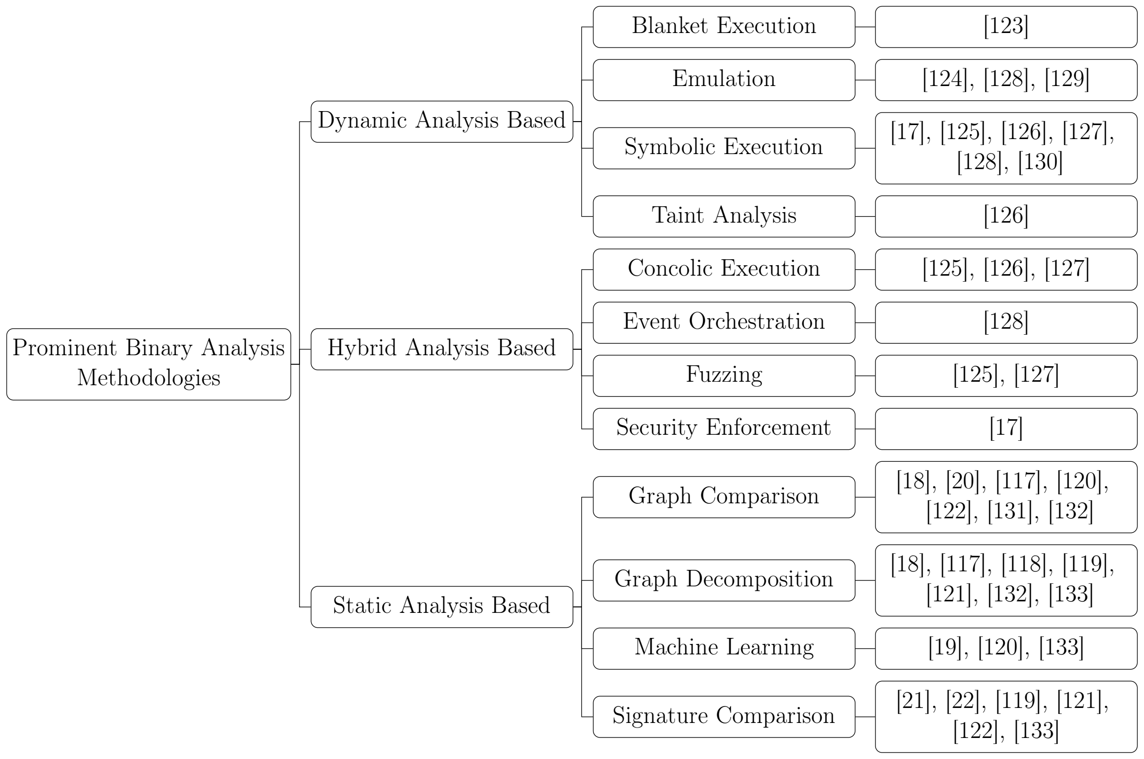


Figure 8: Prominent Binary Analysis Methodologies

3.1.2.1 Dynamic Analysis

This form of analysis is the process of evaluating a given software application through its execution behavior [134]. This can be achieved by monitoring its behavior on a real or an emulated device. The purpose of such a process is to determine the presence of run-time errors or vulnerabilities. This can be achieved through the use of a set of test cases that are composed of specific inputs. Dynamic analysis can also be used to validate the reachability of known vulnerabilities that have been identified in other manners such as static analysis. General advantages of dynamic analysis include the ability to easily circumvent packed or obfuscated code [128]. Limitations to this process include high computational requirements, access to an emulated or physical device and high coupling with them [20]. Another limitation is its difficulty to be applied on several different CPU architectures. This is due to its use of architecture specific tools that are required in order to execute binaries [20]. Together these problems make it inherently very difficult to apply dynamic analysis on specialized hardware such as embedded devices [128]. Dynamic analysis has previously been used for a wide range of different software evaluation. This can include taint propagation, symbolic execution, unpacking and malware sandboxing [128].

Out of all the different ways dynamic analysis be implemented, three specific approaches stand out in their prevalence [134], namely (i) function call monitoring, (ii) function parameter analysis and (iii) information flow tracking. *Function call monitoring* consists of analyzing the execution behavior of a designated function. This is implemented through the use of hook functions that perform the actual analysis. *Function parameter analysis* focusses on the analysis of the run-time parameter values that are provided and returned from designated functions. Such analysis procedures enable the correlation of function calls based on their pre- and post-execution state changes. *Information flow tracking* analyses specifically focusses on the constant monitoring of noteworthy data in a given system during its execution. Values of interest are labeled and said to be tainted. These are then tracked throughout program execution. Scenarios are used to describe how labels are assigned and propagated throughout variable interactions and function execution.

Blanket Execution: It is a concept specifically introduced by Egele *et al.* [123]. It consists of executing a given function several times, with varying inputs, in order to attempt to acquire full basic block execution coverage. This is in order to ensure that the assembly-level features cover all parts of the function.

Emulation: This category englobes all techniques that leverage hardware device emulation in order to perform their dynamic analysis procedures. This can be achieved in many ways including complete hardware emulation, hardware over-approximation and firmware adaptation [128]. Other well-known dynamic analysis techniques, such as symbolic execution or taint analysis, can then be performed on the emulated device.

Symbolic Execution: It consists of the observation and analysis of program behavior when subjected to a certain set of inputs. This can be achieved by leveraging a program interpreter that simulates the execution behavior of a given target. This can be performed both on a binary level (e.g., S2E [135]), or on a higher level representation (e.g., KLEE [136]).

Taint Analysis: This form of analysis involves the tracking of information flow throughout a program. Information flows from one entity to another when it derives information from the another [137]. This allows for the tracking of untrusted information throughout an entire system to see which of its components are tainted.

3.1.2.2 Hybrid Analysis

It is an analysis technique that combines both static and dynamic analysis approaches [138]. Common approach combinations include fuzzing and selective concolic execution [126, 127], and emulation with on device execution [128]. Hybrid analysis can enable the combination of the benefits of all underlying techniques, which results in more complex vulnerability identification [127]. However, hybrid approaches can often result in worse performance, both in efficiency and positive rates, when compared to a purely static one [138].

Concolic Execution: This hybrid technique combines both symbolic execution along with concrete input execution. This input value determines the behavior that the symbolic execution process will have by forcing it to take a certain execution path [125].

Event Orchestration: It is used to help mitigate limitations that arise during embedded device emulation due to specialized I/O peripherals. It combines on device specialized behavior execution and external firmware emulation [128].

Fuzzing: This technique is used to identify program vulnerabilities or crashes. This is done by executing the program while providing it with random or invalid inputs and observing its behavior [125].

Security Enforcement: Technique used to identify paths to operations that require a privileged state in order to execute. Can be implemented in the form of a security policy. It is a definition of what actions are allowed within a given system that enable it to be secure. These can be leveraged in order to define program behavioral constraints that can be used to analyze a given programs privileged behavior [17].

3.1.2.3 Static analysis

This form of analysis can be defined as the inspection of a given program through the direct analysis of its code [134]. As such, in contrast to dynamic or hybrid analysis, no program execution ever takes place. Such procedures can be applied to two distinct program formats. The first is directly upon an applications source code, and the second is on its reverse engineered binary contents. The purpose of such analysis procedures range greatly and include property checking [139], bug identification [120], compiler identification [140] and authorship attribution [141, 142]. General advantages of performing static analysis on source code are its ease of execution from a usability standpoint, as well as direct knowledge of where issues reside in the source code of an application.

This leads to use cases such as non-security specialized individuals, that can easily perform static analysis utilizing a known framework to quickly identify and repair basic security issues. Advantages of statically analyzing binaries are that source code is not always available, making binary analysis indispensable [139]. However, static analysis methodologies suffer from several drawbacks that need to be addressed. Reasonable false positive and negative rates must be ensured. Too many false positives will flood the results and obscurer valid ones, resulting in a large waste of time for the analyst [139]. Too many false negatives will greatly reduce the applications utility and cause it to be discarded. Furthermore, it is often the case that precision time tradeoffs need to be performed in order to provide realistic execution times. Static analysis methodologies are generally designed for a specific set of languages, architectures and vulnerability types. This results in very specialized applications, many of which have to be used in conjunction in order to achieve well-rounded results. Static analysis of binaries comes with its own specific set of difficulties. Reverse engineering is often a difficult process that can yield varying results [123]. This can be due to architecture, obfuscation, encryption and variation due to utilized compilers and options. It is often difficult to understand the original programs semantics based on its binary and link identified vulnerabilities back to source code locations in order to resolve them [139]. Many analysis procedures also suffer from limitations such as function inlining and instruction re-ordering. Static analysis procedures can be implemented in several different ways. One of the most traditional means is through direct semantic analysis. This is done by looking for specific functions or patterns that lead to known difficulties such as buffer overflow vulnerabilities [139]. Another commonly used method is the analysis, and comparison of extracted application graphs. These come in a wide variety such as CFGs and are compared using technique like Graph-Edit Distance (GED). This approach enables a multi pronged (e.g., semantic, syntactic and structural) analysis of a given program.

Graph Comparison: This methodology encapsulates any analysis technique that fundamentally relies on the comparison of graphs extracted from binaries or functions. Examples of such graphs include function CFGs, Data Flow Graphs (DFGs), and Dependence Graphs (DGs). Figure 9 gives a general taxonomy of the different identified approaches.

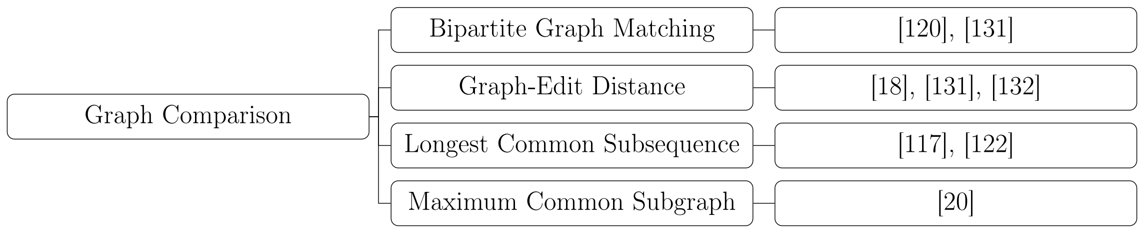


Figure 9: Graph Comparison Taxonomy

Graph Decomposition: This category represents analysis techniques that further decompose graphs that have been extracted from binaries or functions into more granular constructs. These graph components are then used as the primary basis for comparison. Figure 10 gives a general taxonomy of the different identified approaches.

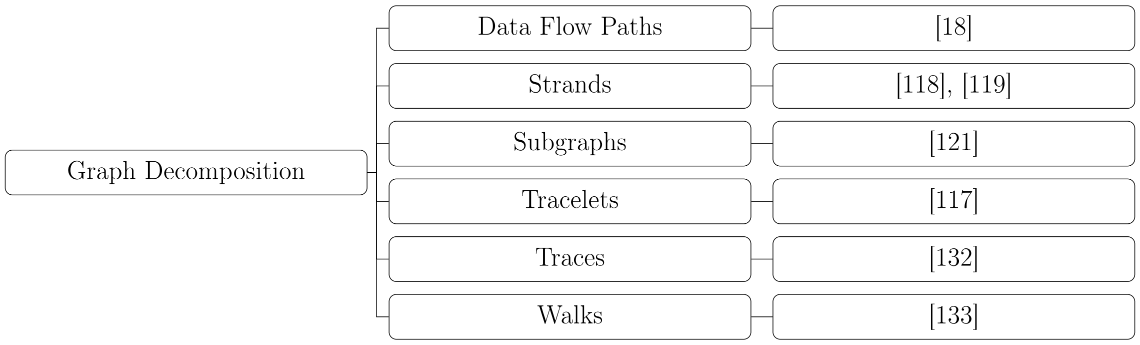


Figure 10: Graph Decomposition Taxonomy

Machine Learning: These approaches rely on learning based algorithms to perform the core of the code similarity comparison process. These can leverage technologies such as neural networks and unsupervised learning. Figure 11 gives a general taxonomy of the different identified approaches.

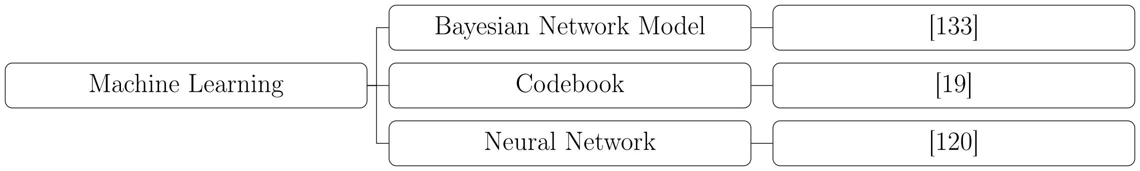


Figure 11: Machine Learning Taxonomy

Signature Comparison: This approach extracts features from binary functions or graphs in order to create a single signature that is representative of the given input. These are then used to perform actions such as similarity comparison and indexing. Figure 12 gives a general taxonomy of the different identified approaches. Furthermore, these approaches often rely on the use of features. These aim at providing a higher level representation of underlying binaries in order to facilitate tasks such as storage and comparison. Figure 13 provides a taxonomy of commonly used types of feature.

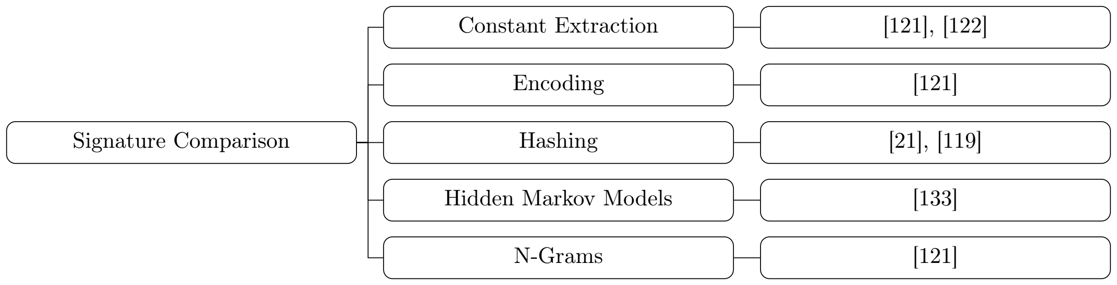


Figure 12: Signature Comparison Taxonomy

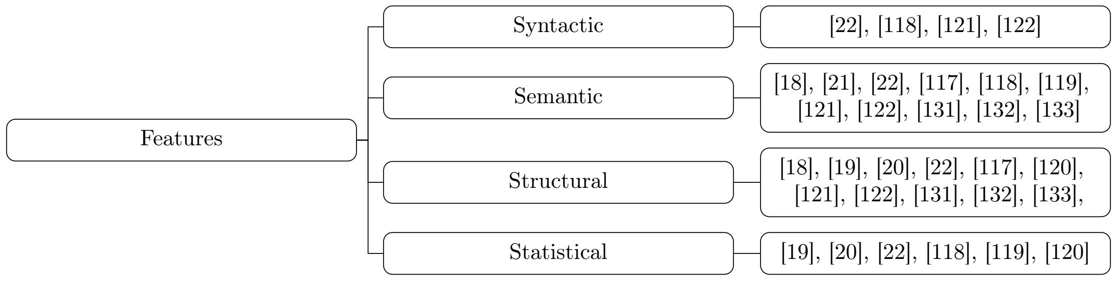


Figure 13: Feature Taxonomy

3.2 Prominent Binary Analysis Frameworks

3.2.1 Search Engine

Here, we discuss all frameworks that can be classified within the search engine category. Subsequently, these are compared to one another and criticized. Throughout this entire process, an emphasis is placed on identifying aspects that are specifically relevant to smart grid IED firmware analysis.

3.2.1.1 Frameworks

An overview of each works purpose, important contributions, techniques, and datasets are discussed in the following sections.

3.2.1.1.1 BinSequence

Huang *et al.* propose [122] BinSequence a code-reuse detection framework. There are several proposed use cases for such an application including bug search, patch analysis, and malware analysis. This framework operates on a binary functional level. It is composed of two main phases namely (i) filtration, and (ii) fuzzy CFG comparison. The filtration phase proposes two filters that operate based on function basic block count similarity, and function fingerprint similarity using MinHashing [122]. The comparison process leverages the LCS algorithm to match a given functions basic blocks with another's. Matches are then further inspected using neighborhood exploration.

Longest Common Subsequence: For a given binary, it is first reverse engineered using IDA Pro where its CFG and mnemonics are extracted. All mnemonics operands are normalized based on their type (e.g., registers, memory references and constants). When performing a comparison between a target and reference function, the targets longest path is first extracted. The reference function is then explored using LCS in order to identify which of its paths is most similar to the targets longest one. Once a match is found, it is further extended by exploring the surrounding structure of each of block in the paths. Should these be similar enough, they are matched together. This procedure is described as a greedy, localized fuzzy matching approach. Basic block similarity is computed using a score that is generated off its normalized instructions.

Dataset Acquisition: Several pre-compiled open-source or publicly available projects are first manually acquired. These are then used to perform all experiments. Libraries include zlib, libpng and libvpx. Furthermore, a few malware samples are acquired, namely Citadel and Zeus. These datasets are specifically designed to demonstrate the frameworks capabilities. As such, the large primary dataset is used to evaluate the frameworks efficiency and scalability.

Bug Search: Within the current context of vulnerability discovery, the proposed bug search use case is the most interesting. This is due to its direct relevance to the current IED vulnerability identification scenario. A single heap-based buffer overflow vulnerability that is known to be present inside several older versions of Firefox [143] is first identified. This vulnerability is then leveraged as a target function in an attempt to prove that the framework can correctly identify it inside vulnerable and non-vulnerable versions of Firefox. The methodology clearly outperform competitors such as BinDiff [144], Diaphora [145] and PatchDiff2 [146] when they are set to perform the same task.

3.2.1.1.2 BinSlayer

Bourquin *et al.* [131] propose BinSlayer a binary similarity calculation methodology. The ultimate goal for such an application is the determination of malware structural similarity. It performs function CFG comparison by formulating this task as a bipartite graph matching problem. As such, at a high-level this process can be seen as the computation of the GED metric between two given CFGs. More concretely, this is achieved by leveraging the methodology proposed by BinDiff

[147] alongside the Hungarian algorithm. Effectively, this enables the identification of the best basic block matches within the two given CFGs. By utilizing such an approach, several challenges that BinDiff has, such as incomplete matches, are mitigated.

Bipartite Graph Matching: BinSlayers graph matching procedure takes place in four main phases. Given two CFGs, BinDiff is first applied in order to get an initial set of basic block matches. This procedure generally results in a set of unmatched blocks. The Hungarian algorithm is then leveraged with the purpose of identifying further missed matches in the remaining set. A newly proposed validation algorithm is then leveraged. Its purpose is to scrutinize the results of the Hungarian matching process. This is achieved by adjusting the utilized edit costs in order to provide more accurate matchings based on structural differences. The final GED comparison metric is then calculated with all results and provided as output.

Dataset Acquisition: BinSlayers dataset is composed of two libraries, namely GNU coreutils along with libxml2. It is not explicitly stated which compilers or optimization flags are used. However, given the nature of the work, it is safe to assume that the GCC compiler for the X86 based architecture are leveraged. The chosen libraries are utilized to perform benchmark evaluations along with comparisons with the BinDiff framework.

Evaluation: One of the main motivations for this work is a novel way to automatically classify malware samples. Unfortunately, no malware related experiments are performed. However, one of the presented observations that relates to graph-isomorphism is noteworthy in respect to this effort. It is stated that library versions with large version gaps (e.g., comparing coreutils 6.10, and 8.19) generally result with less accurate comparison results then those with small ones (e.g., comparing coreutils 8.15, and 8.19) [131].

3.2.1.1.3 BLEX

Egele *et al.* [123] propose blanket execution (BLEX), a dynamic-analysis-based binary function semantics similarity matching framework. It operates on the basic assumption that similar code will have similar execution behavior given similar inputs. Thus, the framework executes two given binaries functions with identically random inputs. During this process it records seven assembly-level features for each function (e.g., heap values, stack values and external function calls). Each functions features are then summed using a weighted Jaccard algorithm and compared. Through this methodology, it is possible reach a conclusion on the two functions semantic similarities and arrive at a final comparison score.

Blanket Execution: When performing blanket execution for a given binary, all of its functions must first be extracted. Each of these is then executed multiple times using randomly generated inputs in a consistently generated execution environment. It is important to note that upon the re-execution of a function, the flow is restarted from the beginning of non-executed blocks and not the original entry point. This is in order to maximize basic block execution coverage and ensures a more efficient way of fully traversing it. BLEX proposes the concept of an execution environment, which is composed of the set of values that are present in all registers and memory locations. In order to receive consistent and comparable results, this environment is identical for both compared functions and reset randomly before every execution. The blanket execution process for a given function results in a vector that is composed of all feature and environment values. Two vectors that have the same environment can be compared using a normalized weighted sum of the Jaccard indices in order to acquire a similarity metric. Effectively, this form of comparison can be viewed as correlating the side effects of function execution.

Dataset Acquisition: BLEX leverages a dataset solely composed of a single library, specifically GNU coreutils. It is diversified using three different compilers (e.g., GCC, ICC, Clang) along with all applicable optimization flags (O0 to O3). Furthermore, the debug symbols are preserved and leveraged as a ground truth in order to validate a given match in the evaluation phase. The weights for each feature are determined using Weka [148] along with a Support Vector Machine (SVM). A total of $9,000$ randomly selected functions from the acquired dataset are leveraged to calculate the weights.

Search Engine: One of the main use cases BLEX is evaluated upon is its capacity to identify a given function within a large corpus. Although not directly attempting to identify vulnerable functions, it is clear that such a use case is quite similar. Functions from the proposed dataset are randomly chosen to act as query functions. This process results in an accuracy of about 77%.

3.2.1.1.4 DiscovRE

Eschweiler *et al.* [20] present DiscovRE, a cross-platform framework that can be used to identify critical security vulnerabilities inside binaries and firmware. Its primary use case dictates that an analyst, whom is in possession of a known vulnerability, wishes to see if it is present in a given binary executable. Fundamentally, these algorithms perform a similarity metric computation based of the comparison of a target function against any number of reference functions CFGs. In order to do so, the framework first extracts a set of lightweight features that are leveraged through the K-Nearest Neighbor (KNN) algorithm as a numeric pre-filtration step. Remaining CFGs are then compared based on their structural similarity using the Maximum Common Subgraph (MCS) isomorphism algorithm.

Filtration Phases: A set of syntactic features that are to be leveraged during the numeric filtration phase are first identified. The purpose of this process is to improve the efficiency of the overall comparison. The robustness of the selected features is then evaluated using the pearson product-moment correlation coefficient. This is to only preserve those that have a small compilation variation and a large value range. This is to ensure proper feature distinction. Several tests are then performed in order to determine the most viable value of k to utilize as a filtration threshold in the KNN algorithm [20]. All functions in the proposed dataset are used to perform these operations. The structural similarity comparison process can then take place on all remaining candidates. This phase leverages CFGs whose nodes have been enriched with additional robust features. These are compared using a modified MCS algorithm that takes into account the basic block features. So as to prevent exponential run-time, the execution of this algorithm is terminated after a set number of iterations. This value is determined experimentally based on their datasets convergence speed.

Dataset Acquisition: Within DiscovRE, an impressive dataset in respect to platform and compiler selection is accumulated. As such, the GCC, Clang, ICC and Visual Studio compilers along with over $1,700$ unique options are leveraged. Resulting binaries are furthermore compatible with the ARM, MIPS and X86 architectures. A set of seven open-source libraries, which contain a total of over $31,000$ unique functions, are also manually collected. Furthermore, two firmware images are acquired, namely DD-WRT router and Netgear ReadyNAS.

Cross-Architecture Bug Search: A relevant use case is proposed where it is attempted to identify vulnerabilities inside binaries. Specifically, the the Heartbleed and POODLE vulnerabilities from openssl are first extracted. Following that, these vulnerabilities are identified within the context of the open-source library dataset and the two acquired firmware images. The efficiency of this process heavily depends on the size of the firmware images, and functions that are compared.

3.2.1.1.5 Esh

David *et al.* [118] propose Esh, a multi-platform binary function statistical code syntax similarity comparison framework. The core of this work is built upon the principal of similarity by composition. That means one code fragment is similar to another if it is composed of parts of the other. As such, each basic block in a given CFG is decomposed into small comparable segments named strands. Syntactic static similarity comparison is then applied on these code segments to infer, when aggregated together, possible functions they correlate to.

Statistical Syntactical Similarity: For a given binary, it is first reverse engineered using IDA Pro and all of its procedures (e.g., uninterpreted functions) are extracted. The assembly code in each one is then transformed into LLVM IR and into a CFG using BoogieIVL. From there, strands can be extracted from each basic block. This is achieved by using program slicing. It involves identifying and extracting all code that manipulates each variable inside the basic block. Strands can then be compared on a semantic level using program verification. It utilizes the proposed Local Evidence Scores (LES) that measures the probability that a given strand composes another. As such, it measures the significance of a given pair of strands. The resulting significant pairings between two functions strands can then be summed together using the proposed Global Evidence of Similarity (GES) metric. This enables the computation of a statistical similarity of two functions purely based on their syntax.

Dataset Acquisition: The proposed target database is composed of solely two open-source libraries, namely openssl and coreutils. These are manually acquired and compiled using GCC, Clang and ICC. This is done with the O2 and O3 optimization flags. Surprisingly, although capable of supporting multiple architectures, all binaries are only compiled for the X64 architecture. This procedure results in a total of 1,500 procedures within the dataset. A few other binaries are mentioned without any details such as ffmpeg, wget and ws-snmp. This corpus is utilized to perform all evaluations.

Vulnerability Analysis: A very relevant evaluation scenario is proposed in which the framework is leveraged in order to perform vulnerable function identification within a set of binaries. This scenario specifically considers 8 distinct vulnerabilities, including Heartbleed, Shellshock and Venom. These are each searched for and identified within the target database.

3.2.1.1.6 Gemini

Xu *et al.* [19] present Gemini, a multi-platform binary code similarity detection engine. The most novel aspect of the work revolves around the use of graph embedding in conjunction with siamese neural networks networks. The purpose of which is to encode or embed a given Attributed CFG (ACFG) into a single feature vector. This encoding scheme provides the distinct advantage of ease when it comes to comparing ACFGs using a generic distance metric.

Siamese Neural Network Embedding: For a given binary, it is first reverse engineered to extract each functions CFG. From there, it is further processed by extracting a few basic block level features and utilizing them to generate an ACFG, a CFG whose nodes are replaced with the contexts of each blocks feature vector. This structure is further encoded into a single feature vector using a siamese neural networks network that utilizes a modified graph embedding scheme. The resulting vectors of both a target and reference function are then compared using the cosine distance metric. It is a difficult task deploying such a solution in the training of the neural networks due to limited quantities of ground truth data. In order to circumvent this problem, a default policy training scheme is proposed. In order to enlarge the set of valid matches, several different function compilation variations are generated, using different optimization flags and architectures. Each resulting binary function variation is marked as a valid association. By grouping functions as such, it is possible to make the neural network associate functions that have the same source code but are compiled for different architectures.

Dataset Acquisition: Three distinct datasets that are each used for different purposes are proposed. The first is used to train the siamese neural networks network. Two version of openssl are first manually acquired and compiled. This is done using GCC along with all optimization flags and platform variations. The second dataset consists of 8,128 usable firmware images from 26 different manufacturers. The third dataset consists of 154 vulnerable functions. These datasets are utilized to perform all framework use cases and evaluations. It is interesting to note that all data present in the firmware and vulnerability datasets are taken from [120].

Vulnerability Analysis: The most relevant use case involves the identification of two CVEs, namely CVE-2015-1791 and CVE-2014-3508, within the firmware dataset. These are specifically selected in order to promote comparability with other work such as [120]. In order to increase the accuracy of all results, additional retraining of the model is proposed. This is applied respectively for each vulnerability and takes about 3 iterations.

3.2.1.1.7 Genius

Feng *et al.* [120] propose Genius a multi-platform binary bug search engine. It performs binary comparison on a functional level leveraging their CFGs. The novelty of this work revolves around the proposed methodology to transform a given binary functions CFG into a single compressed feature vector. More formally, this is stated as generating an ACFG embedding. From there, these vectors can be compared using a distance metric to provide a final similarity score.

Codebook Embedding: Given a binary, it is first reverse engineered and all of its functions CFGs are extracted. Next, a total of 7 basic block level statistical and structural features are identified. These are then leveraged to generate a functions ACFG. This ACFG is then converted to a single vector value that can be efficiently indexed and searched for. In order to generate the embeddings, a codebook based approach is leveraged. Here similar ACFGs are grouped using an unsupervised machine learning algorithm, namely spectral clustering. The centroids of these clusters, being the single most representative point for each on, can be extracted and inserted into a codebook. Then for each function that is to be indexed, it is first compared to all ACFGs that are present in the codebook using the bipartite graph matching algorithm. This results in a set of similarity measurements that together create the given ACFGs representative feature vector. This result can then be indexed in a hash table using Locality-Sensitive Hashing (LSH). When a new target is presented for which matches are required, its embedding is first generated. Given its representative high-level feature vector, all indexed references that are nearby can be identified. A final similarity value is generated using the Euclidian distance between the two feature vectors.

Dataset Acquisition: Three substantial datasets that are leveraged for all evaluations and use cases are presented. The first dataset consists of two versions of BusyBox and openssl. These are each manually acquired and compiled using GCC and Clang with all optimization flags and supported architectures. This resulted in a total of 568,134 functions. The second dataset is composed of two firmware images, namely DD-WRT router and ReadyNAS. The third is composed of firmware that is acquire from previous work [89, 124] and with the use of a web crawler. This results in a total of 33,045 firmware images, 8,126 of which could be unpacked. The last dataset is composed of 154 vulnerable functions, identified by their CVEs, all extracted from openssl.

Vulnerability Analysis: A very interesting firmware vulnerability identification case study composed of two distinct scenarios is performed. The first scenario utilizes the second dataset that is composed of a large number of general-purpose firmware images. This experiment specifically focuses on the identification of two known vulnerabilities within the openssl library, namely CVE-2015-1791 and CVE-2014-3508. The second scenario utilizes all vulnerabilities present in the fourth dataset and attempts to identify them in the two firmware images present in the second dataset. Through this procedure, it is possible to compare the frameworks accuracy and efficiency to previous work such as [20].

3.2.1.1.8 GitZ

David *et al.* [119] present GitZ a stripped binary procedure similarity computation engine. The primary aim is to surpass the common problem of binary variation due to utilized compilers, platforms and optimization flags. Furthermore, this is done in a scalable manner that attempts to minimize false positive rates. A technique that can identify semantically equivalent code that differs syntactically is proposed. The main contribution is the use of a procedure normalization process that is dubbed re-optimization.

Canonical Normalized Form: For a given binary, its contents are first reverse engineered and all procedures are dissembled. These are then lifted into LLVM IR and decomposed into strands. This is the same concept of strands that is proposed by [118] in Esh. Strands are then transformed in a two-phased normalization process. When applied to all strands in a procedure, the resulting output is said to be in canonical normalized form. First, they are canonicalized by re-optimizing or normalizing certain instruction patterns to simpler more comparable forms. Second, registry and memory references are normalized. The given procedures resulting canonicalized strands are then hashed and stored to increase search and comparison time. Each strands relevance to a given procedure is assessed using a statistical model and removed if it is deemed non-representative. Strands are deemed relevant if they are representative of the source code and not the compiler or platform. This is determined using a strand common appearance rate within a large body of sample binary procedures called a global context. Two processed procedures are compared by determining the number of common strands they share.

Dataset Acquisition: A total of 9 open-source projects including openssl and coreutils are identified. A tool that can automatically compile each of them using variations such as compiler, architecture and optimization flags is then created. As such, binaries for the X86 and ARM platforms are generated using GCC, Clang, and ICC with all optimization flags. This procedure results in a corpus of about $500,000$ procedures. Out of these, $1,000$ procedures are randomly selected to compose the global context.

Vulnerability Analysis: A relevant evaluation use case is proposed where GitZ is leveraged as a vulnerability search engine. In order to be comparable to previous work Esh, the same set of 8 vulnerabilities are utilized while adding a single new one. The entire corpus of $500,000$ is used for these procedures. This results in the identification of all true positives along with zero false positives.

3.2.1.1.9 Multi-MH

Pewny *et al.* [21] develop a binary function level vulnerability identification methodology named Multi-MH. One of the core objectives is to create a architecture agnostic approach. The main contributions are the creation of a CFG basic block MinHashing algorithm and a metric that enables the comparison process. This is achieved by deriving basic block I/O behavior, which enables the creation of specific function and bug semantic signatures. These can then be used to match similar functions together.

Min Hashing: When given a binary to analyze, it is first reverse engineered and disassembled. The CFGs of all functions it contains are then extracted. Each CFGs basic blocks code is transformed into an equivalent Vex-IR format. In order to simplify and normalize the resulting IR, the Z3 theorem prover is leveraged. Sample normalized basic block semantics are then determined by observing I/O behavior utilizing random concrete input values. This results in a set of I/O pairs for each basic block. The ensemble of these for a given CFG is considered its signature. The MinHashing algorithm is further applied to each I/O pair in order to create LSH hashes that can be stored and used as lookup indexes. When performing a function comparison, its LSH signature is first computed and all basic block matches are identified. These are then further compared using the proposed Best-Hit-Broadening (BHB) comparison algorithm. This process computes the similarity of two function CFGs based on their matched basic blocks.

Dataset Acquisition: Approximately 60 open-source libraries are acquired and compiled. These include BusyBox, openssl and coreutils. They are all cross compiled for all supported architecture (e.g., X86, ARM and MIPS) using GCC and Clang with all optimization flags. Another set of approximately 9 publicly available firmware images from DD-WRT, NetGear, SerComm and MikroTik are also identified. This number is never clearly stated and is derived from firmware used for experiments.

Bug Search: A set of experiments are performed in which known vulnerabilities or backdoors are identified within firmware or libraries. All identified vulnerabilities are based off of known CVEs such as Heartbleed. Several vulnerability experiments proposed here, for instance those involving the DD-WRT and NetGear firmware, are often repeated in other work [19, 20, 120] as capability benchmarks.

3.2.1.1.10 Rendezvous

Khoo *et al.* present Rendezvous [121] a binary function based similarity search engine. It places a specific emphasis on creating an algorithm that takes efficiency and precision into account. In order to do so, given a reverse engineered binary, a set of feature extraction procedures are applied to each of its functions in order extract relevant and comparable information. These features are composed of function instruction mnemonic n-grams, CFGs and constants. This information is further leveraged to create search terms that enable the identification of other functions that share similar features.

Abstractions: In order to generate a n-gram model for a given function, all of its mnemonics are extracted, encoded and joined into a single block. Here, n-grams are preferred to n-perms due to the added accuracy. The function CFG is then extracted and from it k sub-graphs are chosen to be preserved. This is achieved by treating a given node as a root, and traversing it until k nodes have been seen. The results are then encoded into vectors of length k^2. Constants are leveraged due to the fact that they do not vary during the compilation process. Both integer and string constants are extracted and preserved. Similar function searching is performed by encoding the abstraction features into a 32-bit integer that is used as a bloom filter index.

Evaluation: Rendezvous dataset is composed of two libraries namely coreutils ($1,205$ functions), and the GNU C library or more formally glibc ($2,706$ functions). All binaries are compiled using GCC, and Clang with the O1 and O2 optimization flags. A few possible use cases for their framework are mentioned, namely copyright enforcement, code-reuse detection and vulnerability identification. However, none of the performed experiments cover these use cases. Libraries such as glibc are leveraged to measure performance, and coreutils to perform precision and recall metric calculation.

3.2.1.1.11 TRACY

David *et al.* [117] introduce a syntax based stripped binary function matching algorithm named TRACY. The primary uniqueness of this work revolves around the introduction of tracelets. These are the series of instructions taken from k continuous basic blocks that belong to a given functions CFG. It is a string that is composed of the concatenation of all instructions present in a limited CFG path whose branch or jump instructions have been replaced. Tracelets are comparable using a proposed edit distance metric called rewrite rules. That is the number of rewrites necessary to convert a given tracelet into another.

Tracelets: Given a binary, all of its functions CFGs are extracted. From there, a small normalization step is performed. This involves tasks such as replacing function offsets with their names, and replacing global memory references with concrete values. From this CFG k tracelets are extracted. These are further normalized to remove misleading variation in the form of registers, memory locations and function names. These are then compared for equivalence by leveraging the LCS algorithm in conjunction with constraint-solving techniques. A final comparison score is then calculated based off the obtained tracelet matches.

Dataset Acquisition: A basic dataset is created out of a few open-source tools and libraries such as coreutils and wget. These are compiled using GCC with the O2 optimization flag. This results in a dataset of about a million functions. This dataset is leveraged in order to determine the ideal value of k and to perform a few simple function identification tests.

Vulnerability Analysis: A single simple vulnerability analysis procedure is performed within [117]. A known vulnerability, namely CVE-2010-0624, is first identified. From there, two versions of GNU tar and a single of GNU cpio that suffer from this vulnerability are acquired. These are leveraged to see if the framework can correctly identify vulnerability signatures inside given binaries.

3.2.1.1.12 XMATCH

Feng *et al.* [18] propose a multi-platform binary bug search algorithm named XMATCH. The main contribution revolves around the introduction of a semantic feature called a conditional formula. This methodology can be used to identify bad data dependancies and invalid condition steps within a given function. It does so based on extracted data dependancies and condition checks. With this approach, the aim is to resolve the issues of CFG structural differences, syntactic variation, and vulnerability location dispersal. This is where vulnerabilities can be spread over multiple varying basic blocks due to compiler and platform variation.

Conditional formulas: For a given binary, its CFGs are extracted and its instructions are normalized using LLVM IR. This is achieved by leveraging McSema [149], which has been enhanced to not only support X86 but also MIPS. The next step is to extract the set of conditional formulas that compose a function. These are, at a high level, composed of actions that are executed with given inputs if a set of conditions evaluate to true. The first step in this process is to identify the set of outputs that the function has. These are named action points. Using this information, a use-def chain analysis can be performed to identify all inputs that are linked to a given action point. This results in the creating of a data flow path. Each of these paths is folded in order to create a data-flow equation or an action. Conditions are generated by leveraging a path-slicing algorithm on the identified actions. It identifies all variable comparisons within the given action and formulates them as predicate expressions or conditions. In order to compare two functions, their conditional formulas are leveraged. This is done by representing them as Abstract Syntax Trees (ASTs) and comparing them using GED.

Dataset Acquisition: Two versions of openssl and busybox are manually acquired. They are compiled using GCC and CLANG for the X86 and MIPS architectures. This results in a total of 72 binaries or $216,000$ functions. There is however no mention of utilized optimization flags. Furthermore, the DD-WRT router firmware is acquired and leveraged for experimentation.

Vulnerability Analysis: In order to perform vulnerability identification analyses, a total of 10 known vulnerabilities that are present in compiled libraries are identified. These include the Heartbleed bug. Two analysis scenarios are performed. The first identifies known vulnerabilities inside the acquired DD-WRT router firmware. The second matches the vulnerable functions compiled for X86 and MIPS to ensure cross platform identification capabilities.

3.2.1.2 Comparison

Table 12: Search Engine Comparison

Proposals	Methodology				Features				Feature Levels			Architectures			Compilers			
	Graph Comparison	Graph Decomposition	Machine Learning	Signature	Syntactic	Semantic	Structural	Statistical	Instruction	Basic Block	Function	X86-64	ARM	MIPS	VS	GCC	ICC	Clang
BinArm [150]	●			●	●	●	●	●	●	●	●		●			●		
BinSequence [122]	●				●	●	●		●	●	●	●				●		
BinSlayer [131]	●					●	●			●	●	●				●		
DiscovRE [20]	●			●			●	●	●	●		●	●	●	●			
Esh [118]		●			●	●		●	●			●	●	●		●		●
Gemini [19]			●				●	●	●	●		●	●	●		●		●
Genius [120]			●				●	●	●	●		●	●	●		●		●
GitZ [119]		●				●		●	●			●	●	●		●	●	●
Multi-MH [21]	●					●				●		●	●	●		●		●
Rendezvous [121]				●	●	●	●		●		●	●				●		●
TRACY [117]		●				●	●			●	●	●				●		
XMATCH [18]	●					●	●				●	●		●		●		●

The methodology proposed by BinSequence results in very high accuracy results at 92%. Similarly to DiscovRE, this is coupled with the use of several light-weight filtration phases that are proposed in order to deal with the complexity of function comparison. These enable a scalable and efficient way to compare millions of functions. A similar approach can definitely be leveraged in the context of IED firmware analysis. Several in-depth use cases are proposed. The aforementioned bug search use case is very interesting. Such an evaluation procedure can be leveraged while simply increased its scope. Another interesting use case is proposed as malware analysis. Here BinSequence identifies common functions shared between the Citadel and Zeus malware. A variant of this procedure is the identification of malware, through its function signatures in a given binary, such as an IED firmware. This could help identify malicious content that has been injected within firmware received by manufacturers. BinSequence while being an interesting and powerful algorithm suffers some limitations. Its accuracy is slightly impacted by instruction reordering that can be caused by compiler variation. This is due to its use of LCS during comparison. Due to the chosen graph matching approach, the framework cannot handle function inlining. An example of this is for a given target function, it will be impossible to identify other nested functions it contains.

The most novel contribution of BinSlayer is its introduction of the Hungarian algorithm within the proposed approach. It is used to increase result accuracy and has since been reused in several works [131]. Due to the high accuracy of such an approach, it can be applied to the context of IED vulnerability detection. However, due to scalability concerns, it cannot be directly utilized and should be coupled with other optimizing procedures. The largest limitation of BinSlayer is the use of a graph-isomorphism based algorithm [123]. This is simply due to the fact that several different CFGs can be produced from any given function [120]. This variation is caused by factors such as optimization flag variation, and can result in bad comparison scores when utilizing such approaches. Furthermore, due to the complexity of such algorithms, they cannot be directly applied to large datasets [20].

An important lesson to be learnt from DiscovRE is in its implementation of its simple feature based filtration scheme. It is designed to increase performance while not impacting accuracy. However, Feng *et al.* [120] demonstrate that it instead results in a lower accuracy rate while at the same time increasing false positives, and reducing obfuscation robustness. This clearly demonstrates that the arbitrary use of simple features is not sufficient. It lends credence to the methodology proposed by BinShape [22]. It also demonstrates that it is necessary to perform evaluations to measure the impact of all proposed filtration schemes to ensure they behave as intended. Similar to BinSlayer and many others, DiscovRE leverages a graph-matching-based approach that leads to scalability issues. A specific issue with the use of the MCS algorithm to perform this task is the necessity to preemptively terminate its execution [20]. This is in order to produce realistic comparison times but reduces the accuracy of results. Another unexpected but interesting limitation of DiscovREs approach is its imposed constraint of a minimum function size of five basic blocks [20]. Due to small size of certain CFGs, they have fewer constraints that lead to a much higher chance of false positives [120]. Together these two constraints are interesting since functions must fall into a specific complexity range in order to be fully analyzed. When implementing an IED vulnerability detection engine, such constraints should be avoided. Other general limitations include the inability to handle function inlining, and obfuscation [20]. DiscovRE is able to support multi-platform binaries though the use of syntactical features. This is very interesting and differs from most other work, such as Esh, that leverage IR. When creating their dataset, a very large number of compiler options is leveraged, which results in a total of $1,700$ binaries. However, it is demonstrated that many of these do not produce any significant binary variation [20]. This can be observed through the reduction of the datasets size down to 600 binaries by removing extremely similar or duplicate results. This demonstrates that it is not necessary to leverage all compiler options when generating a dataset, but only the most significant ones.

One very distinguishing aspect of Esh is its use of LLVM IR in order to support multiple platforms. This along with the proposed statistical approach, enables the methodology to overcome compiler variation and small code changes. This same approach unfortunately also introduces a few limitations. Similar to DiscovRE, functions that do not have enough complexity generate very simple strands. These have a high collision chance and thus induce an increased false positive rate [119]. Furthermore, the algorithm cannot handle obfuscated code and does not scale to millions of function comparisons [119]. Due to its reliance on the use of a program verifier, its average runtime is 15.3 hours [118]. Due to this distinct limitation, it renders this aspect of this methodology unusable in the given context. This is due to the base requirement that an analyst needs to be able to identify vulnerabilities within a reasonable time period. An interesting observation that can be made about the proposed dataset is that it is limited to binaries that are only compiled for the X86 architecture [118]. This is despite the frameworks multi-platform support. As such, it is not fully demonstrated its multi-platform capabilities. In addition, the proposed dataset is small in scope and only contains $1,500$ functions.

Gemini is the only identified work that leverages neural networks to perform similar function identification. The authors claim that such an approach enables higher accuracy, embedding efficiency, and accelerated offline training when compared to previous work Genius [19]. Although this is the case, other methodologies do outperform it. For example, Gemini has an average accuracy of about 84% and BinSequences is 92%. This is partially due to its set of simple features. The training time required to incorporate a single new vulnerability into the model can take over 12 hours. This is very large when compared to DiscovRE that has a normalized average indexing time of 0.14 minutes. Although novel, such weaknesses in the methodology make it completely unrealistic to be utilized in the context of IED vulnerability identification due to realistic usability constraints. A large aspect of this work rests upon the shoulders of previous work, Genius. This is due to the reuse of the same general methodology, features, data structures (e.g., ACFG), and datasets.

The main contribution of the use of codebooks by Genius is very unique. No other surveyed work has outlined a methodology of this sort. A large improvement of this approach, over others such as DiscovRE, is an increase in overall function search time. The proposed methodology circumvents its compiler optimization variation limitation by simply generating all optimization permutations. A large general-purpose firmware dataset is presented that is created by combining those created by Pewny *et al.* [21] and Eschweiler *et al.* [20]. An issue with Genius is the length of time it takes to create a new codebook. This process can take more than a week to perform [19]. As in Gemini, when a new vulnerability is discovered, it must be quickly indexed into the reference repository in order to perform timely IED firmware evaluation. Moreover, the quality of all generated codebooks heavily relies on the unsupervised machine learning algorithm [19]. It is therefore unclear if an initial dataset consisting of only vulnerable functions would perform properly in this scenario. This is because the quantity of vulnerable functions is relatively small (e.g., hundreds to a few thousand). If this is the case, the codebook approach becomes unusable in a purely vulnerability identification based scenario. What further puts weight on such a claim are the results of their firmware vulnerability analysis process results with a precision of 28% [19]. This is extremely low. Similar to DiscovRE, there is an imposed restriction on functions that have at least five basic blocks [120].

GitZ is the successor of Esh and thus reuses large aspects of it. This includes the entire IR lifting process, instruction normalization procedure, and utilized datasets (e.g., libraries and vulnerabilities). It is important to note that the proposed strand scheme is limited to the context of a given basic block [119]. As such, a very questionable aspect of the methodology revolves around the use of the MD5 hash to index strands. Due to the nature of this hashing algorithm, this process will only index identical strands into the same bucket. This results in the overall procedure being extremely affected by any processed strand variation. This can be clearly observed through the results of the similarity score experiments where there are only exact matches or none at all. However, a distinct yet obvious advantage of such an exact matching scheme is that there are virtually no false positives due to the MD5 algorithms weak collision resistance property. In the context of IED vulnerability detection, there is the need for not only exact matches, but granular ones. That way implementation limitations that cause accuracy decreases, due to slight variations, will not be automatically discarded and can be considered. It would be interesting to investigate the effects of varying the chosen hashing algorithm for another such as LSH. Furthermore, relative strand position within the graph is not taken into account. This is because the methodologies similarity metric only measures the ratio of shared strands two functions have. A more important issue is the frameworks average runtime of 1.8 seconds [119]. This is relatively slow when compared to other proposed solutions such as DiscovRE or Genius and will clearly not scale to millions of functions.

Two distinct advantages that the approach Multi-MH proposes are resistance to instruction reordering and function inlining. This is respectively due to its use of IO pairs and basic blocks as the basis of function comparison [21]. This point is particularly noteworthy since no other work identified boasts such capabilities. Just like BinSequence, Multi-MH explicitly discards all obfuscated binaries from its analysis process. Unfortunately, like many other approaches, its general comparison procedure has scalability issues [20]. This is primarily due to its use of the MinHash algorithm in the offline preparation aspect of the work. As such, it can take one month to index and search a stock Android image that contains about 1.4 million basic blocks [20]. This is a large detriment to this approach given the complexity and quantity of IED firmware that can be potentially analyzed. The methodologies online comparison is still expensive due to its graph matching approach [120]. There is also a large number of false positives generated with this approach [21]. David *et al.*

[118] state that this is potentially due to limitations in its sampling approach and that basic block frequencies are not taken into account.

According to Feng *et al.* [120] Rendezvous has two main disadvantages. These lie first in its use of n-gram features that can be limited by instruction re-ordering that does not change overall program semantics. The second is in its function decomposition approach where search accuracy will be reduced for two very similar CFGs that have a small number of varying edges or structural differences. Due to this slight variations, the algorithm will decompose the graphs into very different subgraphs. Furthermore, the methodology is unable to handle function inlining since it operates at an assembly function level [151]. Another set of limitations is introduced by the use of k-graphs [121]. This is because the efficiency and accuracy of the approach is greatly tied to the chosen value of k. If a small value of k is utilized, this will result in very similar graphs. The utilized value of k is also determined with a small dataset. It is likely that this chosen value will have to be recalculated if different or more libraries are utilized. Rendezvous evaluation dataset is very small and consists of only two libraries with a total of $3,911$ functions. As such, performed evaluations do not represent a thorough scalability evaluation.

A unique aspect of TRACY is its ability to handle function inlining in certain limited cases [117]. It is also interesting to note that the proposed approach can hypothetically handle cross-domain assignment but at a much higher cost [120]. Bad results are produced if there are less then 100 basic blocks in the evaluated functions [117]. This is because smaller functions have a higher rate of common tracelets and such similarity often leads to false positives. Many vulnerable functions do fall into this category; simply discarding them is not an option. Due to this its approach is inapplicable in the context of IED vulnerable function identification. Furthermore, TRACY is unable to handle variations generated by compilers such as equivalent instructions and instruction reordering [117].

The main contribution of conditional formulas by XMATCH is very unique. No other surveyed work has outlined a methodology of this sort. Their use of an intermediate representation is good way to enable a consistent way to create conditional formulas. However, their choice of McSema is questionable. It does not support multiple architectures or the entire X86 instruction set [149]. This is not a significant issue, but it does slightly reduce XMATCHs overall accuracy. Another reduction of accuracy lies in its use of single loop unrolling when performing data flow analysis. This however is a tradeoff to increase the overall efficiency of the process [18]. XMATCHs is unable to handle function-inlining and vulnerability dispersal across several different functions [18]. Finally, as many other approaches that leverage graph comparison algorithms, the use of GED is considered expensive.

3.2.2 Malware Library Reuse

Here, we discuss all frameworks that can be classified within the malware library reuse category. Subsequently, these are compared to one another and criticized. Throughout this entire process, an emphasis is placed on identifying aspects that are specifically relevant to smart grid IED firmware analysis.

3.2.2.1 Frameworks

An overview of each works purpose, important contributions, techniques, and datasets are discussed in the following sections.

3.2.2.1.1 BinShape

Shirani *et al.* [22] propose BinShape, a library function reuse identification. Relevant use cases for such an algorithm include reused library vulnerability identification and malware detection. For any given pair of functions, the proposed comparison scheme is composed of both an offline and online phase. During the offline stage, reference libraries are processed and indexed. For each function they contain, a specific set of features is identified and a unique signature is generated. The online phase operates on a given target function, whose features are extracted and compared to those stored using a modified B+ tree. A two-phased filtration scheme, based on the number of basic blocks and the instruction count, is used to speed up this process.

Signature Generation and Indexing: The most distinct and novel part of the methodology revolves around the function signature generation process. The first step is to identify and propose a large corpus of heterogeneous function CFG features. These are composed of a wide range of graph, instruction-level, and statistical features. When generating a signature for any given function, the values of each feature is extracted. The resulting importance and relevance of these values is ranked using Mutual Information (MI). Finally, in order to minimize the signature size, a decision tree classifier is leveraged. The purpose of which is to isolate and preserve only the best features for a given function. This effectively results in the signature that is indexed inside a proposed data structure dubbed a B++ tree. It is composed of a set of B+ trees, one for each proposed feature. Each of these B+ trees contains a set of references to indexed functions whose final signature contains its designated feature. This enables an efficient indexing and detection time of log(n) complexity.

Dataset Acquisition: In order to perform all evaluations and use cases, a large number of open-source C/C++ libraries and tools are manually acquired. This set includes libraries such as 7zip, expat, and libcurl. These are compiled using GCC, and Visual Studio with all four optimization levels. Furthermore, the ground truth is generated by leveraging Program Debug Databases (PDBs), which contain debugging information. This process results in a corpus of over 1 million functions. The leaked Zeus malware source code is also acquired and compiled.

Malware Detection: BinShape performs an interesting use case where library reuse detection is attempted within the context of a real-world malware sample. Using publicly available information, the framework identifies known libraries that the Zeus malware leverages. Finally, these identified libraries are acquired and compiled in order to validate that they are in fact present in the Zeus malware.

3.2.2.1.2 FOSSIL

Alrabaee *et al.* [133] propose FOSSIL, a framework for identifying Free Open-Source Software (FOSS) library reuse in malware. As such, if the FOSS libraries that malware use can be identified, it is possible to infer some of the malware's functionality. However, this process can be rendered difficult due to binary obfuscation. The authors primary purpose is to find a novel way of defeating traditional obfuscation techniques that many other algorithms suffer from. This includes binary variations such as utilized compiler and optimization flags. It is done in three main levels of binary function analysis that are (i) syntactical feature extraction, (ii) semantic feature extraction, and (iii) behavioral feature extraction. The results from each of these are then merged together using a Bayesian Network model and are used to identify possible FOSS package reuse.

Detection Methodology: Given a reverse engineered binary, all of its instructions are first normalized in order to remove meaningless differences such a memory references. A Hidden Markov Model (HMM) is then applied on extracted syntactical features. These are taken from a set of opcodes contained in the function that have been selected based on their prevalence using mutual information. The functions CFG is then broken down into walks, which contain semantic relation data. This information is extracted using a neighborhood hash graph kernel. Finally, the behavior of instructions in a function is quantized using the z-score metric. It is applied to each normalized instructions frequency distribution in relation to the rest of the function. A Bayesian Network (BN) is used to coordinate the execution of these three phases. It can measure all acquire information and identify correlations between them.

Dataset Acquisition: They first create a dataset composed of 52 FOSS packages that are acquired on Github. Before being selected, these packages are ranked based on popularity and malware reuse in order to identify the most pertinent ones. These are manually acquired and compiled using GCC and Visual Studio resulting in a total of $340,071$ functions. Several malware samples are obtained, disassembled and de-obfuscated. This includes copies of the Zeus, Citadel, Stuxnet, Flame and Duqu malware. Furthermore, a large set of $5,000$ unanalyzed malware ranging from 12 families is acquired.

Malware FOSS Analysis: The objective here is to identify FOSS libraries that malware samples use. The set of reverse engineered malware is first tested with FOSSIL to act as a ground truth evaluation. This is possible since during the manual analysis process all utilized FOSS packages are identified and technical reports about them are publicly available. All other acquired general malware are then also evaluated as a scalability benchmark, and to gather a large amount of results.

3.2.2.1.3 SIGMA

Alrabaee *et al.* [132] design a malware binary function reuse identification algorithm named SIGMA. The main contribution is the introduction of the Semantic Integrated Graph (SIG), a data-structure that merges the traditional CFG, Register Flow Graph (RFG), and Function Call Graph (FCG) together. This enables a new level of semantic description of a given binary function. In order to compare them, they are decomposed into sets of short paths named traces.

Semantic Integrated Graph: The methodology consists of the reverse engineering followed by the disassembly of a given binary. From there, for each identified function its CFG, RFG and FCG are extracted. Each of these is further enhanced with graph specific features. As such, CFGs are converted to structural Information CFGs (iCFG) through the addition of basic block colors that represent instruction derived functionality. Then RFGs are converted to Merged RFGs (mRFG) which introduce new classes and merge equivalent ones. Finally, FCG are augmented into Colored FCG (cFCG) to distinguish internal from external system calls. These three augment graphs are then merged together into a single SIG. Short paths within this structure are then annotated as traces. These are then used in conjunction with the GED algorithm in order to perform comparisons.

Malware Detection: SIGMA proposes an interesting but simple function identification use case. Samples of the Zeus and Citadel malware are first acquired. Then the RC4 function contained within them is then identified using the proposed methodology. This function is specifically chosen since it is known to be present in both malware instances.

3.2.2.2 Comparison

Table 13: Malware Library Reuse Comparison

Proposals	Methodology				Features				Feature Levels			Architectures			Compilers			
	Graph Comparison	Graph Decomposition	Machine Learning	Signature	Syntactic	Semantic	Structural	Statistical	Instruction	Basic Block	Function	X86-64	ARM	MIPS	VS	GCC	ICC	Clang
BinShape [22]				●	●	●	●	●			●	●			●	●		
FOSSIL [133]	●		●			●	●		●		●	●			●	●		
SIGMA [132]	●					●	●			●	●	●			●	●		

An interesting aspect of BinShape is its capability to resist binary variation in the form of utilized compilers and obfuscation techniques. However, it is not designed for vulnerability detection and as such is not directly relevant to IED vulnerability identification. Where BinShape is of particular interest is in its concept of function shape. This is demonstrated through its general methodology in respect to its feature selection and ranking procedures. BinShape proposes a large set of lightweight features that cover graph, instruction and statistical levels all of which are of particular interest. It is unclear which of these are the most relevant in the context of IED firmware and vulnerabilities. They should all be evaluated for this new context, possibly using a similar approach. This work demonstrates a distinct advantage of such a methodology in that it increases the overall robustness of the solution by reducing the chance of signature collisions [22]. Another interesting and relevant result is that statistical and graph based features are the most relevant for open-source libraries [22]. This is because most IED vulnerabilities can be extracted from open-source libraries. Finally, it would be interesting to apply the proposed malware use case to IED firmware in order to attempt malware identification within them.

The most relevant aspect of FOSSIL for this given context is its methodology with regards to open-source software. Using domain relevant FOSS libraries as a basis for its evaluation and analysis use cases is extremely interesting. A similar approach can be leveraged to identify IED firmware specific FOSS libraries that contain vulnerabilities.

SIGMA supports a limited form of obfuscation resistance in the form of register variation and instruction reordering [132]. A large set of features are proposed and are used. These can definitely be useful in IED firmware evaluation. However, a large quantity of function features are required in order to train the model. A very small set of evaluations are proposed, which could be improved. The authors state themselves that future work should expand upon this with function fragments, compiler variation and optimization flag variation tests [132]. The introduction of a SIG is very novel. Unfortunately, given this data structures complexity, performing SIG comparison is not a scalable solution and can thus not be applied to IED firmware vulnerability identification.

3.2.3 Vulnerability Discovery

Here, we discuss all frameworks that can be classified within the vulnerability discovery category. Subsequently, these are compared to one another and criticized. Throughout this entire process, an emphasis is placed on identifying aspects that are specifically relevant to smart grid IED firmware analysis.

3.2.3.1 Frameworks

An overview of each works purpose, important contributions, techniques, and datasets are discussed in the following sections.

3.2.3.1.1 AEG

Avgerinos *et al.* [125] present AEG, an automatic program exploit identification and generation algorithm. It distinguishes itself by being the first methodology that can perform such a functionality [126]. This is implemented by performing a hybrid analysis that leverages both static source analysis, symbolic and concolic execution. This multi-pronged approach is chosen for its increase in the scalability of the bug identification and exploitation procedure [125]. AEG primarily focusses on memory based exploits in the form of control-flow hijacking due to the fact that they can be modeled with relative ease [17]. This model can then be reused in the context of many different analysis.

Exploit Generation: When performing analyses, a copy of the binaries source code is mandatory. As such, it must be acquired either though public releases, private channels, or reverse engineering. The main justification for a source code based approach is that it provides a higher level overview of the application, which facilitates automatic processing [125]. Symbolic or concolic execution is then performed on the source code in order to identify vulnerable execution paths (e.g., buffer overflow). AEG then attempts to solve the symbolic values in order to identify concrete ones that can trigger the bug. Dynamic analysis is then leveraged to acquire memory layout information that is used to inject the malicious shellcode. The resulting exploit is then executed and validated to have worked.

Binary Acquisition: A relatively small set of 14 open-source applications is leveraged in order to validate the approach. These are all manually downloaded and compiled for the X86 architecture using GCC. These were specifically chosen for they contained known vulnerabilities. These known weaknesses are then used to validate the frameworks capabilities.

Vulnerability Discovery: The focus is primarily on the identification of known vulnerabilities. This process results in the identification of 16 vulnerabilities, two of which being zero-day exploits. Although these results seem promising, Shoshitaishvili *et al.* [152] present another picture. As a benchmark, AEG is reimplemented and tested against 126 binaries. Out of these only 4 exploits are generated [125].

3.2.3.1.2 Avatar

Zaddach *et al.* [128] propose Avatar, a hybrid embedded system security testing framework. The methodology primarily distinguishes itself from other work by introducing a hybrid analysis approach. This implies that the firmware is emulated, while all peripheral I/O operations are forwarded to the original device. The emulation is achieved using a generic CPU emulator. A proposed event-based arbitration framework is responsible to orchestrate all communication, and suspend emulation while I/O operations are taking place. This is performed by injecting a software proxy into the embedded device. Such a process enables the capability to perform complex dynamic analysis of firmware such as vulnerability discovery and hardcoded backdoor detection.

Event Orchestration: On the embedded device side, either a debugging interface (such as JTAG) or a custom injected in-memory proxy stub is used to communicate with the framework. It is entirely event based; all analysis takes place using exchanged device and emulation data streams. All emulation is powered by S2E [135], a symbolic execution engine. Depending on the firmware, Avatar can be run in several different modes. Full-separation mode is used for unknown firmware images. Here, the firmware is fully run on the emulator while all memory related operations are executed on the physical device. The context switching mode primarily runs the firmware on the native device. It is only when a point of interest is reached that the execution context is switched over to the emulator for further analysis.

Firmware Acquisition: Avatar is meant to be a flexible analysis framework that can cover a large variety of embedded devices. As such, the authors acquire and evaluate three distinctly different devices, namely a GSM feature phone, a hard disk boot loader, and a wireless sensor node. Each device must be properly set up to function with the framework. This involves tasks such as documentation identification, debugger presence determination, and stub injection. This process takes a large amount of time and expertise. Due to this a much larger scale analysis process would be difficult to perform.

Vulnerability Analysis: Avatar is intended to be used as a powerful multipurpose framework that can perform tasks such as malware analysis, vulnerability discovery and backdoor detection. The hard disk boot loader is analyzed using full-separation mode followed by S2E. By analyzing its communication protocol, it is possible to identify a contained hidden backdoor. As for the ZigBee-capable wireless sensor node, its firmware is modified in order to inject a buffer overflow vulnerability. Afterwards, by symbolically exploring all execution paths, it is possible to identify it. Finally, for the Motorola C118 phone, there is a specific focus on its SMS decoding capabilities with its network stack. It is run in context-switching mode where emulation only takes over once a SMS is received. Unfortunately, performing symbolic execution on such a process results in a state-explosion condition. Due to this, no final results are presented.

3.2.3.1.3 Driller

Stephens *et al.* [127] present Driller, a hybrid analysis-based binary vulnerability discovery framework. It has two noteworthy contributions. First, the concept of a component graph is proposed. In such a structure, each node represents a code junction where specific user input checks are performed. Second, the concept of selective concolic execution, which is a combination of traditional fuzzing and concolic execution techniques, is presented. Fuzzing is first used in order to identify all code sections that are responsible to validate user input. This results in the creation of the component graph. A limited concolic execution is then performed by only considering inputs for the identified components. In doing so, the otherwise present path explosion weakness is mitigated [127].

Selective Concolic Execution: When a given binary is to be analyzed, Driller first leverages its AFL fuzzing engine. The fuzzer is run until it fails to report any new state transitions [152]. The framework then transition over to its symbolic tracer (e.g., angr [153]) and provides it with all unique paths the fuzzing engine identified. This has the added benefit of limiting the state explosion problem [127]. Its purpose is then to discover new inputs that open up other paths. When such inputs are discovered, the fuzzing engine takes over again and leverages them. This two phased process is repeated until an input that crashes the application is discovered.

Binary Acquisition: In order to validate Drillers capabilities, the DARPA Cyber Grand Challenge dataset is employed. It is composed of a total of 131 different binaries, each of which has one unknown vulnerability. A total of 5 of these are discarded due to the fact that they perform inter-binary communication, which is out of scope of the work. As such, the remaining 126 binaries are leveraged to perform all evaluations.

Vulnerability Discovery: The main focus of evaluations is on memory corruption vulnerabilities. The framework successfully identifies a total of 77 out of 131 possible issues that are present within the dataset. This result outperforms the baseline fuzzer that is provided within the context of the DARPA Cyber Grand Challenge by 12%. Surprisingly, the introduction of selective concolic execution is only responsible for identifying 9 vulnerabilities; all others where discovered with the traditional fuzzing approach.

3.2.3.1.4 FIE

Davidson *et al.* [130] propose a symbolic execution engine, named FIE, whose purpose is to perform simple firmware security analysis. It distinguishes itself by specifically targeting firmware for the $16 - bit$ MSP430 micro-controller. It is commonly used in security critical environments such as in Point of Sale (PoS) devices and smoke detectors. FIE specifically focusses on the detection of resource access bugs and vulnerabilities. FIE builds upon KLEE [136], a powerful symbolic execution engine that generates test cases and leverages LLVM [154]. The dynamic analysis scheme is such that all execution paths present in a given firmware image are executed and evaluated.

Symbolic Execution: The size of the firmware present on MSP430 micro-controller is relatively small. As such, traditional scalability issues related to symbolic analysis do not come into play. This is what enables the evaluation of all firmware execution paths. The problem of infinite execution loops and state explosion is handled via the use of a technique called state pruning. This is where previously visited states have been marked as such and are ignored when they are revisited. A technique that aims at enabling further analysis called memory smudging is proposed and leveraged. It allows for the overriding of counter values in order to explore and analyze states that are not normally accessible in exchange for an increase in false positives. Furthermore, the methodology introduces an easy way to specify memory layouts, interruption handlers and peripheral accesses within the context of different MSP430 models. This enables the execution of interrupts at any given execution point. A default configuration is provided that can handle models of unknown behavior.

Firmware Acquisition: MSP430 micro-controllers are RISC-based devices. Their firmware is written in a domain specific variation of C. As such, FIE specifically targets small firmware whose code is available. In this context firmware is acquired from four different sources. Publicly available PoS card reader and USB driver firmware is first manually acquired. Furthermore, Github is searched, both through manual inspection and automatic tag searching, for firmware related to the designated micro-controller. The last code source utilized is taken from Contiki, an operating system in the form of a library that is commonly used in micro-controllers. All acquired code has to be validated for compatibility with the MSP430. It is then compiled into LLVM bytecode that is leveraged to perform the symbolic analysis process.

Vulnerability Analysis: FIE specifically focusses on the detection of memory safety issues (e.g., buffer overflows, out-of-bounds memory access), and peripheral errors (e.g., write to read only memory). This is done by identifying locations where unvalidated or unsanitized inputs have control over the behavior of the application. Generally, such inputs cause the evaluated firmware to crash or code injection should an attacker craft a specific payload.

3.2.3.1.5 Firmalice

Shoshitaishvili *et al.* [17] propose Firmalice, a symbolic execution based embedded firmware vulnerability identification framework that leverages Angr [153]. Specifically, it targets the identification of authentication bypass vulnerabilities and backdoors. For a given firmware image, a security policy that indicates the location of privileged operations is required; that is sections of code that normally require authentication in order to execute. Given a firmware image along with its associated security policy, Firmalice transforms the given binary into an Intermediate Representation (IR) [155] and performs static analysis in order to identify critical authentication sections called privileged program points. These points can be further expanded to limited firmware slices. These are code segments that contain the privileged operations of interest. These firmware slices are then symbolically executed in order to see if there are constraint violations present.

Symbolic Execution: In order to optimize the symbolic execution analysis aspect of the work, the entire firmware image is not executed. Only the relevant subsets of the image are considered for execution. As such, privileged program points specified by the given security policy are leveraged along with a Program Dependency Graph (PDG) in order to create a firmware slice that ranges from the entry point to the privileged point. The symbolic execution of these slices can then take place. The methodology uses symbolic states to track execution progress and identify when it reaches a privileged state, which is when a privileged point is executed. Symbolic summaries are used to take function side effects into account and thus perform a more through analysis.

The core concept behind the authentication bypass detection assumes that an attacker can provide a certain input to the system that will result in an unauthorized privileged state. This is achieved using constraint solvability, or the capacity to properly concretize the required input to bypass the privilege mechanism. This is done by considering all ASCII interrupt values as user input. Furthermore, all outputted data is considered to have been exposed to the attacker.

Firmware Acquisition: The firmware of three different devices is acquired, namely the Schneider ION 8600 smart meter, the 3S Vision N5072 CCTV camera and the Dell 1130n Laser Mono Printer. This demonstrates that Firmalice can operate on a large range of embedded devices. The authors consider the process of acquiring more firmware outside of the scope of the work [17]. It is also infeasible to perform an analysis process on a much larger dataset given the complexity of creating the device specific security policy.

Vulnerability Analysis: This work primarily targets the identification of authentication bypasses and backdoors. This occurs when a user is allowed to perform certain actions that require access rights they do not possess. Through the use of Firmalice, backdoors within each of the proposed firmware images are identified. It is important to note that all identified vulnerabilities where already publicly known. As such, no new problems are identified using this methodology.

3.2.3.1.6 Mayhem

Cha *et al.* [126] present Mayhem, an automatic binary exploit identification and generation framework. Its main contribution is the establishment of a symbolic execution index-based memory model. It introduces the concept of symbolic read address while enforcing concrete write addresses [127]. This enables a more efficient way of handling symbolic memory loads [126]. Mayhem primarily focusses on memory corruption exploits that can allow for the injection and execution of malicious payloads.

Symbolic Execution: Mayhem has two main components, a Concrete Executor Client (CEC), which performs all taint analysis operations and a Symbolic Executor Server (SES) that performs all symbolic execution. The analysis starts locally on the CEC, where a given binary is executed in order to identify tainted branching locations. These are then passed to the SES whose responsibility is to determine if they are accessible. It also uses a SMT solver to fork execution paths and a path selector to prioritize them. Exploits are generated when tainted branch locations are identified. These are called exploitability formulas and are provided to the SMT solver to test their validity.

Binary Acquisition: A small set of 29 publicly available Windows and Linux based binaries are employed to perform all evaluations. These are pre-compiled and for the X86 architecture. Most of these contain known vulnerabilities that allow the authors to validate the performance of the framework.

Vulnerability Discovery: The primary use case for the framework is the identification of known vulnerabilities. This process results in the identification of 29 vulnerabilities, two of which being zero-day exploits. The scope of complex binaries that the methodology is able of analyzing is however quite limited. This is primarily due to its lack of system and library call implementations when performing symbolic execution [126].

3.2.3.2 Comparison

Table 14: Vulnerability Discovery Comparison

Proposals	Methodology								Security Analysis						Architecture				Complexity			Scalability		
	Concolic Execution	Blanket Execution	Emulation	Event Orchestration	Fuzzing	Security Policy	Symbolic Execution	Taint Analysis	Authentication Bypass	Backdoor	Code Injection	Known Vulnerabilities	Memory Safety	Peripheral Errors	ARM	MIPS	X86-64	MSP430	High	Medium	Low	High	Medium	Low
AEG [125]	●						●				●	●	●				●			●			●	
Avatar [128]			●	●			●			●		●			●				●					●
BLEX [123]		●										●					●			●				●
Driller [127]	●				●		●						●				●			●		●		
FIE [130]							●						●					●		●			●	
Firmalice [17]							●		●	●					●	●	●		●					●
Mayhem [126]	●						●	●			●	●	●				●			●			●	

At a high level, the approach proposed by AEG seems extremely interesting for IED firmware analysis. However, in practice it is hardly suitable. A large concern is the mandatory need for the analyzed binaries source code [125]. This resource is generally not easily accessible; it requires significant effort to reverse engineer and decompile a given firmware image. Furthermore, the proposed symbolic approach can quickly consume all available system memory due to large number of possible execution paths [126]. As other works like Avatar demonstrate, performing dynamic analysis on specialized firmware binaries is very complicated [128]. This will likely complicate the proposed Dynamic Binary Analysis (DBA) phase if applied to the context for IED analysis. Shoshitaishvili *et al.* [152] demonstrate that the approach proposed by AEG is still in its infancy. This is due to its small number of identified exploits when evaluating the framework and its relatively simple identification methodology.

The core and novel approach proposed by Avatar is to circumvent the traditional emulation difficulty of having an accurate model of the analyzed device. This is done by proposing a hybrid methodology that leveraging the original device in conjunction with generic emulation [128].

This does greatly increase the scope of different devices that can be analyzed while unfortunately greatly reducing the overall scalability of the solution [124]. The first and most obvious reason is the requirement to have access to the physical device. This is both financially and spatially disadvantageous. When considering IED analysis, this is a particularly large issue. It is unrealistic, due to financial constraints, that a large number of them can be acquired. The next difficulty is the high-level of manual interaction, and knowledge required to integrate each given device into the analysis suite [124]. This is further complicated by the fact that commercial devices very often do not come with debugging interfaces or they are disabled [128]. This is intentional for security concerns and to make it harder to interact with the device in undesired ways. As such, it cannot be assumed that a given IED will have such an interface present. Yet the injection of a proxy stub, and identification of execution points is a difficult process that requires a lengthy and an iterative approach. The different proposed modes of execution orchestration are quite interesting. Unfortunately, the full-separation mode can be very slow and can execute around five memory accesses operations per second [128]. Just like the other solutions that utilize symbolic execution, Avatar suffers from state explosion when the analyzed firmware is too complex as seen in the final proposed use case [128].

The BLEX framework provides a very interesting way to circumvent the optimization flag limitation that many graph matching approaches suffer from. This is done by observing the side effects of a given function; the idea being that irrelevant of optimization level, both should have very similar execution results [123]. An interesting contribution with the blanket execution scheme is the complete coverage of a given functions execution. The authors claim that this process can be extended to obfuscated code [123]. However, BLEX operates by executing individual functions. This process still requires the identification of firmware and function entry points, which can be greatly hindered by obfuscation. Another possible limitation to the approach is function inlining [21]. An example of this is given two functions with identical behavior; the difference between them being that one has all externally referenced functions inlined. The resulting dynamic execution vectors will not be identical due to features such as external function calls. This makes the comparison and vulnerable function signature generation procedure far more complicated. Proposed experiments yield accuracy results of about 77% for function identification. This result can be possibly explained by need for more comprehensive assembly-level features. The utilized evaluation dataset is compromised of $195,560$ functions, all of which are derived from a single library. When compared to other work such as Firmadyne, such a dataset is very small. As such, this allows the framework to sidestep some difficulties, such as scalability and efficiency, that other methodologies face. If BLEX is to be applied on a larger scale, particularly with more than the use of a single library, the feature weights will have to be recalculated. This is due to the fact that they are not representative of a large enough variation of functions to be accurate at a larger scale.

One of the claims the authors of Driller present is that the proposed selective concolic execution approach is able to find more complex bugs than other approaches [127]. However, despite being true, its use provided a relatively small benefit (e.g., 9 out of 77 identified vulnerabilities). Clearly more work is needed on this front. Furthermore, the use of AFL in such a combination is known to pose many challenges [156]. The proposed dataset although quite large and interesting, is unfortunately not applicable to the current context of IED analysis.

The FIE framework is an extremely specialized. It specifically focusses on MSP430 family firmware that has been compiled with msp430-gcc, one of three possible compilers [130]. Due to this choice of full execution path analysis, the methodology is limited to very simple and small firmware images. A large one will cause the symbolic execution process to fail due to a state explosion condition. As such, the chosen approach, assuming it is adapted to other contexts such as IED firmware, has very little to no potential of being usable. The firmware collection process is interesting in the fact that it is partially automated. Certain relevant tags are automatically searched for on Github in order to discover potential candidates and compile them if they meet certain requirements such as the presence of a Makefile.

An interesting aspect of Firmalice is its use of static analysis in order to identify areas of interest on which to perform symbolic execution [17]. Firmalice is very distinct from other solutions in that it performs symbolic execution in order to identify authentication bypass vulnerabilities. Such a security verification procedure comes however at a very high cost [129]. Creating a device specific security policy requires significant device knowledge and a large amount of general expertise. This is due to the fact that identifying code sections that require authentication do not contain any distinct logical features [17]. This requires knowledge into how the given device is supposed to operate. Furthermore, obfuscation can pose a great challenge in identifying these critical locations. As such, the creation of these is very time-consuming and effectively renders the entire solution extremely difficult to scale. Thus, this method of analysis is not particularly feasible when put in the context of a large-scale IED security analysis process. Although state pruning is a good preventive approach, it does not completely prevent the issue of state explosion when utilizing this methodology [127]. Another unique aspect of this work is the utilization of intermediate representation. This effectively allows for the abstraction of the underlying CPU architecture and perform multi-platform symbolic execution analysis. Intermediate representation is a very interesting solution to keep in mind for an multi-architecture based IED firmware analysis solution.

Similarly to AEG, it is difficult to implement the approach presented by Mayhem within the context of IED firmware analysis. Its requirement to implement a large set of system calls in order to perform symbolic analysis is likely to be very challenging for specialized embedded devices. Furthermore, its dynamic taint analysis procedure is also likely to be difficult for such devices as shown in [128]. Concolic execution can also quickly lead to state explosion due to execution path forking [127]. Mayhem proposes an interesting solution to this in the form of checkpoints and path prioritization. This is however not suitable in all cases [126].

Through the general results and problems that each of these solutions have, it is possible to make several observations. These are important and should be taken into consideration. As can be seen through Firmalice, FIE and Avatar, any form of dynamic analysis is a difficult evaluation method to perform in scalable manner. This is due to the possibly large amount of manual preparation and domain knowledge required to perform said task. It is very difficult to simultaneously perform it on a variety of different platforms. A distinct advantage Firmadyne has over all others, and thus is an important point to consider, is its level of automation. This can be seen through the way its datasets and evaluation process dwarfs all other solutions. A scalable methodology needs to be considered right from the start of the project as can be seen in FIE. This is even if domain specific characteristics might not require such a consideration. It is important to rigorously determine the values of feature weights both through the methodology and the quantity of value determining data as can be seem within BLEX. The combination of different traditional approaches can yield some very interesting results as demonstrated by Avatar and Firmalice. Moreover, this work demonstrates that it is important to limit required device access, knowledge, and manual intervention in the overall analysis process.

3.2.4 Web Interface Analysis

Here, we discuss all frameworks that can be classified within the web interface analysis category. Subsequently, these are compared to one another and criticized. Throughout this entire process, an emphasis is placed on identifying aspects that are specifically relevant to smart grid IED firmware analysis.

3.2.4.1 Frameworks

An overview of each works purpose, important contributions, techniques, and datasets are discussed in the following sections.

3.2.4.1.1 Automated Firmware Analysis

Costin *et al.* [129] perform a large-scale embedded firmware analysis. The methodology specifically focus on the web front-end interfaces these devices expose. This area is of particular interest due to the considerable number of vulnerabilities they generally contain and the otherwise general focus on communication protocols. The proposals main contribution lies in the combination of existing techniques into a new framework and leveraging it to perform a large-scale analysis.

Emulation: When given a firmware image, its file system is identified and extracted. Within it all web server binaries, configuration files, setting specifications and source code (e.g., HTML, CGI, etc.) are obtained. These are then all loaded into QEMU [157] using a generic kernel and file system. A plethora of known web interface security analysis frameworks, such as Metasploit [158] and Nessus [159], are then applied in order to perform a vulnerability assessment. This provides the obvious benefit of a very easily extensible analysis suite.

Firmware Acquisition: The presented dataset is quite large and is composed of a total of $1,925$ successfully unpacked firmware images. These are all acquired from the public Internet and can be classified as Commercial-Off-The-Shelf (COTS) device firmware. Device types include routers, modems and CCTV cameras. After evaluating each of them for chroot and web interface emulation capabilities, there remains a total of 246 images that are actually analyzed.

Vulnerability Analysis: The leveraged security analysis frameworks allow for the identification a very large range of vulnerabilities. As such, a total of of $9,271$ possible vulnerabilities are identified within 185 firmware images. Of these, 225 are of high impact.

3.2.4.1.2 Firmadyne

Chen *et al.* [124] propose a methodology named Firmadyne, which is used to evaluate the security of COTS device firmware. There is a specific focus on the automated dynamic analysis of the subset of COTS devices that use Linux-based firmware. This is implemented through the use of software enabled full device system emulation. In order to be able to emulate a wide range of devices, instrumented stock kernels are created. Security evaluation is based off the use of known vulnerabilities that are present in the Metasploit [158] framework.

Emulation: QEMU [157] is utilized to perform all emulation. The firmwares native kernel is not used, but replaced with a generic one that has been specifically configured for a given firmware image. This is because direct emulation using the native kernel will most likely result in a fatal crash due to the unsupported hardware platform. There are three main steps in the process of performing COTS device firmware emulation. Header information in file system binaries is used to identify the specific CPU architecture that is to be emulated. This enables the selection of which pre-compiled kernel should be utilized with QEMU. Once completed, the initial emulation phase can take place. The modified kernel records subsystem interactions for all networking systems. This enables the identification of interfaces that have to be emulated. The actual emulation can then take place with all identified and mocked interfaces.

Firmware Acquisition: A mostly automated process is employed to acquire and properly analyze all firmware images. This involves performing manual or automatic website crawling in order to identify firmware images. This is achieved using Scrapy [160] and covers FTP long with manufacturer websites. These are then downloaded along with any relevant meta-data and automatically extracted by leveraging binwalk. Only resulting kernel and file system data is preserved. Throughout this process several optimizations are performed, such as blacklisting certain file types in order to accelerate the extraction process.

Vulnerability Analysis: Three main phases of dynamic analysis are performed on any given firmware image once it has been successfully emulated. The first is to assess the security of all identified and accessible web pages hosted by the device. This is to identify the presence of common attacks such as buffer overflows and code injection. The second is to evaluate all accessible SNMP MIBs in order to determine if they contain any sensitive information that attackers might target. Finally, the third leverages the Metasploit to perform several known attacks against the device. The result of these is validated through log file analysis.

3.2.4.2 Comparison

Table 15: Web Interface Analysis Comparison

Proposals	Methodology								Security Analysis						Architecture				Complexity			Scalability		
	Concolic Execution	Blanket Execution	Emulation	Event Orchestration	Fuzzing	Security Policy	Symbolic Execution	Taint Analysis	Authentication Bypass	Backdoor	Code Injection	Known Vulnerabilities	Memory Safety	Peripheral Errors	ARM	MIPS	X86-64	MSP430	High	Medium	Low	High	Medium	Low
Automated Analysis [129]			•								•	•			•	•			•			•		
Firmadyne [124]			•								•	•	•		•	•			•			•		

The automated firmware analysis methodology that Costin *et al.* propose has several characteristics that make it appealing to IED web interface analysis. First, the employed high-level of automation would enable a relatively quick processing of a large amount of firmware. Second, it is general enough to be applied in this new context. Furthermore, its utilized security scanners can be extended to cover IED specific vulnerabilities. However, the approach has several points that would have to be improved in order to leverage it. The authors clearly state the issues encountered due to device emulation using a generic kernel. These will most likely be exuberated in the new context. An example of this is frequent crashes caused by unsupported NVRAM memory access. The vulnerability results proposed might be slightly misleading. There is no mention of the possible overlap of results provided by the diverse set of frameworks or an examination of all results to determine which are false positives. If an analyst has to leverage this approach, significant effort would have to be invested in filtering all acquired results in order to determine their validity.

Firmadyne is specifically restricted to the analysis of linux based firmware emulation, and as such do not need access to the physical device. These imposed limitations enable a much higher level of automation and thus scalability. This applies to both the firmware reverse engineering and security analysis processes. This can be clearly observed through its presented analysis of $9,486$ firmware images. This is especially noticeable when compared to other solutions such as Avatar and Firmalice that require significantly more manual intervention. Such an approach is very interesting when considering IED analysis. The downside is that many, but not, all IEDs are Linux-based. Additionally, in order to easily identify the most vulnerabilities as possible, known ones are leveraged though the use of the Metasploit framework. This removes the burden of identifying vulnerabilities and makes the framework both more efficient and usable in a real-world situation. This is due to the fact that very little effort needs to be made on behalf of the firmware analyst when integrating newly discovered vulnerabilities. The use of known vulnerabilities is very interesting for IED analysis since their software is often outdated and rarely or never updated. A clear disadvantage with the emulation approach is the possibility of incompatibility with very specialized and proprietary IED firmware. A clear reason for this is the lack of support for out-of-tree kernel modules present on device file systems. As well, it is unclear if the claim that system-level emulation can handle all interfaces will apply to specialized IED ones. There is a distinct weakness in the process used to discover exposed web interfaces. It is assumed that all web pages are stored statically in the /var/www directory. While this is the default case for some common technologies (e.g., Apache), this location can easily be different based on configuration or utilized servers. As such, it is possible that some web resources can be missed.

3.3 Summary

In conclusion, this chapter provides an overview of the most prominent endeavors in the field of vulnerability identification inside of IED firmware binaries. We propose two large taxonomy groups that aim at classifying both the purposes and methodologies of all examined work. This provides a high-level overview of all prominent and currently available binary analysis approach capabilities along with the set of methodologies and techniques each leverages. Furthermore, it facilitates the classification and comparison of the current state-of-the-art. This results in the identification of four main framework groups namely search engine, malware library reuse, vulnerability discovery, and web interface analysis. Each entry in these categories are compared to one another and criticized. Throughout this entire process, an emphasis is placed on identifying aspects that are specifically relevant to smart grid IED firmware analysis.

Chapter 4

IED Firmware Security Analysis

4.1 Overview

Through the results of the literature review process, it has been concluded that a static analysis based approach is best suited for IED firmware vulnerability identification. This is due to the fact that dynamic analysis approaches are often computationally expensive, and are difficult for firmware images [20]. In addition, these approaches rely on architecture specific tools to run executables, which makes it inherently difficult to support other architectures [20]. This is specifically problematic when it comes to the topic of IED firmware emulation. Given the vast diversity of IED sensors and peripherals, it is impossible to fully emulate them. Furthermore, there exists a more realistic hybrid approach between emulation and on device execution [128]. This solution does not scale. It is due to the physical requirement of having the device along with the time required and complexity of setting it up for such an analysis process.

Due to this choice of approach, an emphasis is placed on the identification, understanding and classification of static analysis based approaches. Machine learning based approaches (e.g., Gemini and Genius) are discarded due to long preparation phases, and requirements for large training sets. It is not realistic to invest several hours of time to retrain a model for every newly introduced vulnerability. Furthermore, there is not a great diversity for each vulnerable function which can cause model training issues. Graph decomposition based approaches (e.g., Esh, GitZ, and TRACY) are also discarded due to several drastic limitations. These include extremely long run-times (e.g., seconds to hours), lack of granular results and minimum function complexity restrictions. Thus, a joint use of both a signature and graph comparison based approaches is the most appropriate for this context. These approaches boast the best compatibility range, indexing times and comparison efficiency. As such, if the speed of signature based detection can be coupled with the accuracy of graph comparison algorithms an efficient, accurate and scalable solution can be produced.

4.2 Approach

At a high level, the purpose of our proposed framework dubbed BinArm [150] is to perform a fuzzy known vulnerability matching in IED firmware. This general process can be subdivided into two main steps namely (i) *offline preparation* and (ii) *online search*. Each of these separate phases can be run independently. However, the results created in the *offline preparation* process are leveraged by the *online search* process. Figure 14 provides a visual overview of this process.

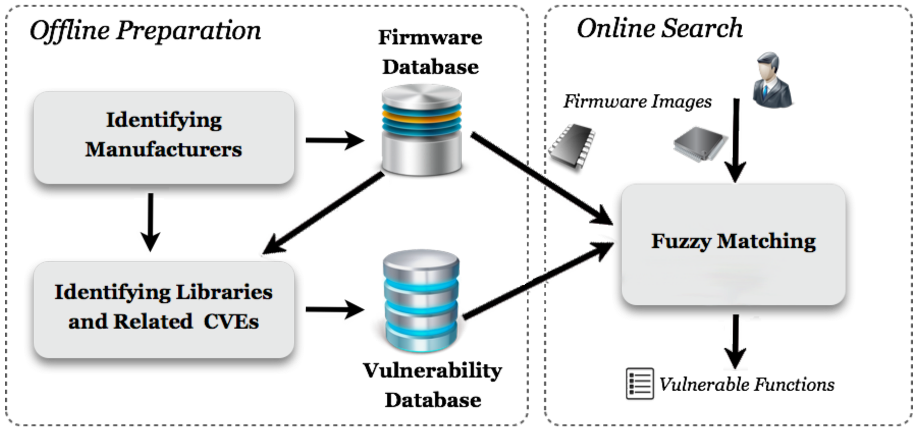


Figure 14: BinArm Approach Overview

In the offline preparation phase, two large repositories of function information are created. For both it is essential to identify a set of prominent smart grid manufacturers. This information is first leveraged in order to identify and acquire a multitude of relevant and publicly available firmware images that they have produced. These are further filtered down by relevance criteria such as CPU architecture. The firmware database is then populated with these acquired, disassembled and processed IED firmware images. The list of smart grid manufacturers is then further leveraged in order to identify sources of utilized open-source software in their released firmware images. This software reuse information can now be combined with known vulnerabilities taken from the CVE [26] database. This allows for the creation of a list of most commonly used vulnerable libraries within the industry. This information can be further expanded into a list of most likely present vulnerable functions. The vulnerability database is thus populated with these divers and vulnerable functions. The creation of these datasets is further discussed in Chapter 5.1.

The online search phase performs the core functionality in BinArm. It performs the actual fuzzy vulnerability identification procedure by leveraging the two previously mentioned sources of information. In order to make this process as efficient as possible, this phase performs a multi-stage detection process that is composed of three main fuzzy matching phases. This is namely a light feature matching phase, followed by a branch feature matching phase and finally a fuzzy control flow graph matching phase. This process results in the accurate matching of known vulnerable functions to potentially vulnerable ones contained inside IED firmware.

4.3 Instruction Normalization

During the offline preparation phase, all firmware binaries are automatically analyzed using IDA Pro [86]. An ARM statement is composed of an optional label, an instruction mnemonic, a destination register and one to three source registers [161] as follows.

{label:*} {opcode {dest {, source1{, source2{, source3}}}}}

When attempting to compare different instructions that are contained inside basic blocks, there are two problematic sources of variance that may hinder this process. First, there is a large number of possible registers that can be used as source and destination in any given statement. An example of this is the fact that there are 16 general-purpose registers in 32-bit ARM [161]. Secondly, it is possible to perform memory address access in order to perform loading and storing operations. Due to this large memory address and register variance, it is possible to have a syntactically different statement that effectively performs the same functionality. In order to resolve this issue, instruction normalization is utilized. This process effectively maps all source and destination types to a single corresponding value. Through this procedure, the problematic sources of variance are removed. IDA Pro already provides a CPU architecture specific normalization scheme for operand types [162]. Thus, this mapping scheme can be leveraged.

However, directly utilizing this scheme would lead to an increase in false positives due to its register grouping being too generic. This is because it regroups several different categories, such as program status and floating point registers, as general registers. This can be resolved by further classifying different register types and assigning them a distinct normalization code. Table 16 describes the additions to the general normalization scheme that is applied to the operands received from IDA Pro disassembly. It is through this process that we are able to create normalized operands in order to perform proper statement matching and comparison.

Table 16: Register Normalization

Register Type	Concerned Registers	Normalized Code
General	a1-a4, r0-r3	100
	v1-v5 , r4-r8	101
	v6, sb, SB, r9	102
	v7, sl, SL, r10	103
	v8, fp, FP, r11	104
	ip, IP, r12	105
	sp, SP, r13	106
	lr, LR, r14	107
	pc, PC, r15	108
Program Status	CPSR	110
	SPSR	111
Floating Point	f0-f7 , F0-F7	120
	s0-s31 , S0-S31	121
	d0-d15 , D0-D15	122
Co-Processors	p0-p15	130
	c0-c15	131
Other	–	140

4.4 Multi-Stage Detection

The detection process represents the core functionality in the fuzzy matching algorithm. It is composed of three main phases that each aim at gradually identifying the best candidate matches for a given function while discarding those that are not. Each phase is gradually more thorough and complex. This is to quickly and cheaply, in terms of computation time, identify a limited set of possible candidates that can be more carefully examined. The three stages are represented by logical components which are: (i) shape-based detection, (ii) branch-based detection, and (iii) fuzzy matching-based detection.

4.4.1 Shape-Based Detection

This subsection was done in conjunction with the author of [22], Paria Shirani. She proposes the concept of a functions shape, which is defined as the set of heterogeneous features that can be extracted from it [22]. By leveraging this work, it is possible to identify a concrete list of usable features. Thus, this phase leverages the most important and relevant of these features in order to assist in pruning the initial search space. This is done by first evaluating our entire general database with respect to the identified features. It is then possible to identify which of them are the most relevant in our given context. From there, only the most important can be used to calculate a relevance threshold that will determine if a given function should be further analyzed.

4.4.1.1 Feature Identification

There are three general types of features that have been identified: (i) instruction-level features, (ii) structural features, and (iii) statistical features.Together, each of these categories provides a well-rounded overview of a certain functions syntax, semantics and CFG. A sample of selected features can be found in [150].

Instruction-level features: This feature represent syntactic and semantic information that a given function contains. An example of this is the number of registers or the number of operands.

Structural features: These are directly tied to the CFG metrics of a function [22, 163]. They represent information that can be extracted from the graphs structural properties. An example of this is a CFGs cyclomatic complexity or the number of nodes it contains.

Statistical features: These represent the semantical information that can be acquired from a given function [22, 164, 165]. For example, a function can be described by the statistical distribution of the instructions it contains [22].

4.4.1.2 Feature Selection

Once all heterogeneous features have been identified, it is necessary to extract the most useful and relevant of them. Utility in this context is the features capability to clearly provide information that enables differentiation between functions. This process is in order to improve the final function shape based detections accuracy and efficiency. The first step in this process is to extract off of these features for all functions in all vulnerable libraries, which includes the entire general and vulnerability database.

In this context, feature relevance is evaluated using a classifier. As such, each functions name is designated as a respective class. The objective of the classifier is to identify which features, when combined, most accurately defines a given function. Given this requirement, the use of a mutual information classifier [166] is ideal. The reason is that this algorithm attempts to find the set of features that optimally characterize a given class. This approach of feature combination is a reason that differentiates this feature selection process from other popular ones such as maximal relevance [166]. The output of the mutual information process is score for each feature (generally in bits of information), which indicates how much its uncertainty is reduced given another feature [167]. Thus, features that are scored higher are more dependent on each other and together they reduce each other's uncertainty. This classification process is implemented using Python leveraging the Science Kit Learn machine learning library [168, 169]. Since our engine performs a coarse detection in the context of the shape-based detector, it is desirable to minimize operation complexity. As such, only the top few ranked feature are chosen namely graph_energy, kurtosis and skewness.

4.4.1.3 Function Matching

In the function shape-based detection phase, it is necessary to discard all reference functions that are not similar enough to a given target one. This is achieved through utilizing a similarity threshold; should any given function surpass the threshold, it shall be discarded. It is thus necessary to define a metric to evaluate the similarity of two given functions. In this scenario, each function is represented by a set of the previously discussed features. There exist a plethora of different metrics that can accomplish this goal [170]. We selected Euclidian distance due to its simplicity, and ease of calculation. Its formula is defined in Equation 1, where p and q are two points in Euclidian space, representing the sets of all chosen heterogeneous function features namely the triple (skewness, kurtosis, graph_energy), for both a given target and reference function. This procedure is formally stated in Definition 1.

$$d_{Euc}(p,q) = \sqrt{\sum_{i=1}^{n} \left|(q_i - p_i)\right|^2} \tag{1}$$

Definition 1. *Let: (i) f_T, f_r be two functions, (ii) p and q be 3-tuples associated with f_T and f_r, (iii) $d(p,q)$ be the Euclidean distance function between p and q, (iv) d the resulting distance, and (v) λ a predefined threshold value. Where: (i) $GSK(f_T) \rightarrow p$ and $GSK(f_r) \rightarrow q$ denotes extracting skewness, kurtosis, and graph_energy from the f_T and f_r functions, (ii) the value of $d \geqslant 0$, and (iii) the value of $\lambda > 0$. Therefore f_r is considered as a candidate function to be matched with f_T, if $d(p,q) \leqslant \lambda$.*

4.4.1.4 Threshold Selection

It is desired to empirically and automatically generate an appropriate threshold value that is to be used to separate valid from invalid candidate functions. This is as opposed to performing manual tests and determining a suitable value through experimentation. This is implemented through the use of the K-Means clustering and the Elbow algorithms in order to group similar functions together. These clusters can then be further analyzed to extract an appropriate threshold value based on their size.

K-Means is an unsupervised machine learning algorithm [171]. It is classified as unsupervised because the dataset that it operates on is unlabeled. The purpose of this algorithm is to automatically partition the provided input data into K clusters based on its features similarity. The provided dataset here is all utilized functions where each respective functions extracted features represents a data point. Each resulting cluster contains an identified centroid point that is conceptually the center of the cluster, and whose feature values represent those of the rest of the cluster [172]. Some limitations of this algorithm are that it can only be applied on numerical data, and that the mean of a given set of data points is defined. This is however not a problem in this particular situation. More formally, K-Means divides p data points into K predefined clusters, $C_1, \ldots, C_K$, who have K centroids, $c_1, \ldots, c_K$. The objective function, defined in Equation 2, is to do so while minimizing the total within-cluster sum of squared errors E [172]. This effectively results in making each cluster as distinct and compact as possible.

$$E = \sum_{i=1}^{k} \sum_{p \in C_i} dist(p, c_i)^2 \tag{2}$$

A known limitation of the K-Means clustering algorithm is the selection of the number of clusters to identify, namely the optimal value of K [173]. The Elbow methods is used in this context due to its ease of use and implementation. Its procedure involves setting the value of $K = 2$, calculating the total within-cluster sum of squared errors, and then repeating the process while incrementing the value of K. Each of these results is then recorded and plotted in a graph. The objective here is to identify the smallest value of K that minimizes the recorded sum of squared errors [174]. Graphically this can be identified by a sudden drop in the error value for a given K followed by a more or less constant decrease afterwards. This drop off point is the ideal value for K and is called the elbow due to its shape. A downside of this approach is that it can sometimes be very difficult or impossible to identify the proper value given that the aforementioned pattern does not manifest. This however is not the case for our results and thus this method is applicable.

From this computed information, it is necessary to derive a proper threshold value. This is done by leveraging the identified centroids in each data cluster. For each point in a given cluster, its distance from its centroid is calculated. Each of these measurements is averaged out to get the average euclidian distance from the centroid for each cluster. Finally, the average of the obtained distances is calculated and set as the threshold value. This procedure is more formally presented in Definition 2. The final value of λ is calculated as shown in Equation 3.

Definition 2. *Let: (i) $\{f_1, f_2, \ldots, f_n\}$ be all the functions in our repository, (ii) p_i and q_i be 3-tuples associated with functions f_i and f_j, (iii) $d(p_i, q_j)$ be the Euclidean distance between f_i and f_j, (iv) K be the number of clusters, (v) $\{c_1, c_2, \ldots, c_n\}$ be all the identified clusters, (vi) n_c represent the number of points in all clusters, and (vii) λ the calculated threshold value. Where: (i) $GSK(f_i) \rightarrow p_i$ and $GSK(f_j) \rightarrow q_j$ denotes extracting skewness, kurtosis, and graph_ energy from the f_i and f_j functions, and (ii) the value of $\lambda > 0$.*

$$\lambda = \frac{1}{K} \times \sum_{c=1}^{K} \left(\frac{\sum_{i=1,j=1}^{n_c} d(p_i, q_j)}{n_c}\right) \tag{3}$$

Figure 16 illustrates the results of the Elbow procedure for our selected feature triple. This process results in a selected value of $K = 11$. Figure 15 gives a graphical overview of the resulting clusters. The threshold value for this process is calculated to be $\lambda = 26.45$. Thus, all reference functions whose Euclidian distance is from the target is more than 26.45 is removed from the comparison set. This selected threshold is further experimentally evaluated in Subsection 5.2.7.

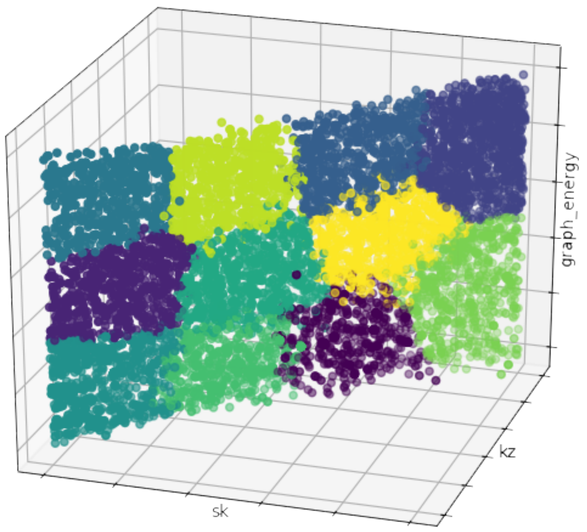
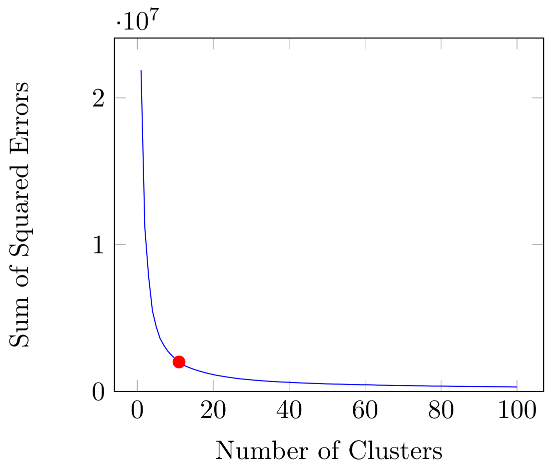
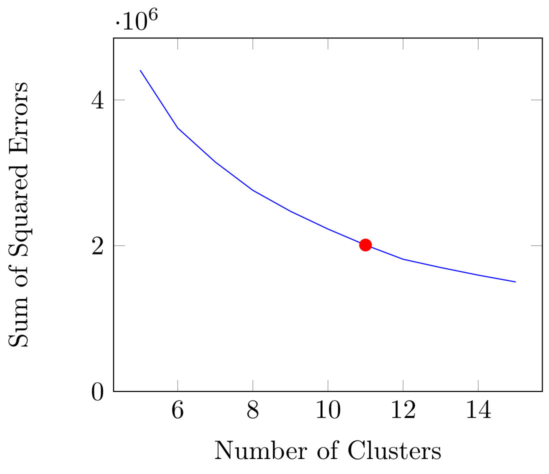


Figure 15: K-Means Results: K=11



(a) Elbow Results



(b) Zoomed Elbow Results

Figure 16: Elbow Results: K=11

4.4.2 Branch-Based Detection

The purpose of this particular phase in the detection process is to further reduce the size of a given comparison set. This is in order to increase the overall performance of the function similarity assessment process. This second phase of the detection algorithm is further used to compare functions from a different informational standpoint from the first. Where the first focusses on function shape, the second branch-based detection phase focusses on function execution paths. In previous work, analyzing execution paths has enabled the capability to identify function vulnerabilities as well as stealthy program attacks [175, 176]. The general idea here is that similar functions will have similar execution paths. Thus, the branch-based detection process takes as inputs the CFGs of a given target and reference function. From there, it leverages a weighted path comparison algorithm to assess the distance between the two graphs. Just as in the shape-based detection phase, if the resulting distance is over a given threshold, the reference function is removed from the pool of possible candidate functions. The following sections discuss the utilized path comparison algorithm, the modifications that we made in order to utilize it, and how it is implemented.

4.4.2.1 Normalized Tree Distance (NTD)

The Normalized Tree Distance (NTD) [23] metric is used to compare the paths of two given trees. The requirement here is that they both have the same topology and number of paths. NTD is a L1 metric that thus satisfies the symmetry and the triangle inequality properties. The output of this comparison process is a distance that ranges from $[0, 1]$. Thus, a distance of 0 implies that two given trees are identical and a distance of 1 indicates complete distinctness.

Originally, this distance metric is proposed to compare two given phylogenetic trees. These trees must have the same topology and same set of n taxonomic group. Additionally, each path has a specific length which is composed of evolutionary rates denoted as weights on each edge that connects any two given vertices. An example of two such comparable phylogenetic trees can be seen in Figure 17.

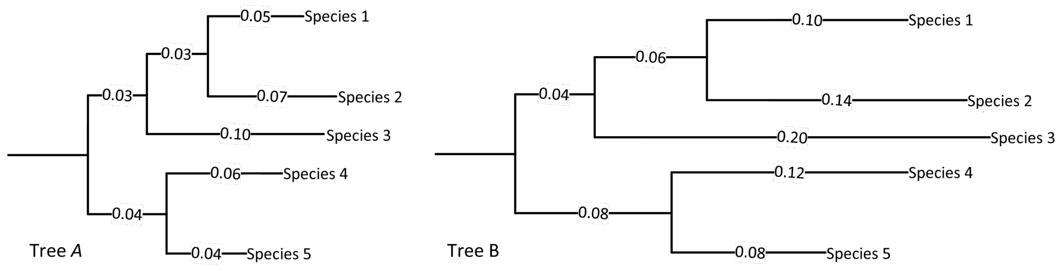


Figure 17: Two NTD Compatible Phylogenetic Trees [23]

More formally the comparison process can be defined as taking two given phylogenetic trees $A = \{a_1, a_2, \ldots, a_N\}$, and $B = \{b_1, b_2, \ldots, b_N\}$ as input. Then they are compared using Equation 4, where N is the number of paths, a_i is the length of path i from tree A, and b_i is the length of path i from tree B.

$$NTD = \frac{1}{2}\left(\sum_{i=1}^{N}\left|\frac{a_i}{\sum_{j=1}^{N} a_j} - \frac{b_i}{\sum_{j=1}^{N} b_j}\right|\right) \tag{4}$$

4.4.2.2 Weighted Normalized Tree Distance (WNTD)

The NTD equation though interesting, is not sufficient if directly applied in our given context. When comparing the execution paths of two given functions CFGs, it is possible, though unlikely, that they both have the exact same number of paths. It is also required to not only take path length into account, but also specific weights associated with each vertex in the given graphs. The final issue is that given the nature of the equation, paths are compared purely based on their indexes. As such, each path i of two given CFGs will be directly compared irrespective of their similarity. An example that illustrates this problem is that given two identical trees, where only the order of their paths are different, their comparison will yield a non-zero comparison result.

Due to these limitations, it is necessary to modify and propose a new version of the equation that is valid for our given problem space. Thus, we propose the Weighted Normalized Tree Distance

(WNTD) metric. The WNTD metric has been refined in conjunction with Paria Shirani. This metric takes as input a weight for each vertex in a given path. Furthermore, given two trees, it is responsible to identify the most similar path matches they contain. Due to this behavior, even if the two trees do not have the same number of paths, or are not ordered the same way, there is still the possibility of identifying similarities. In order to prevent too dissimilar function paths from being associated, a similarity threshold γ is introduced. This handles the case where a match is found but its value is too large making it invalid. Thus, a given match result that has a percent distance of over 50% from the original path is ignored. Experimentation indicates that this threshold value provides good results (Subsection 5.2.7.2).

More formally, the WNTD equation can be defined as taking two given CFGs of functions W and V as input. These are represented as weighted paths such as $W = \{w_1, w_2, \ldots, w_N\}$ and $V = \{v_1, v_2, \ldots, v_N\}$. They are then compared using Equation 5, where N is the number of paths, w_i is the weighted path from CFG W, v_i is the weighted path from CFG V, and v_{BM} is the best match for path w_i when compared to all those present in V. The general formulation of v_{BM} behavior is defined in Equation 6. The exact method for computing the best match for a given path is further discussed in Subsection 4.4.2.3.

$$WNTD = \frac{1}{2}(\sum_{i=1}^{N}\left|\frac{w_i}{\sum_{j=1}^{N}(w_j)} - \frac{v_{BM}}{\sum_{j=1}^{N}(v_j)}\right|) \tag{5}$$

$$v_{BM} = \begin{cases} BestMatch(w_i, \vec{V}) & \text{, if there is any match} \leqslant \delta \\ 0 & \text{, else} \end{cases} \tag{6}$$

A complication arises due to the format of the provided data to the WNTD function. The CFGs that are extracted from functions may be cyclical. As such, it is impossible to assess all the paths that such a CFG contains. Thus, before applying the WNTD equation, it is necessary to process any given graph. This is achieved though the removal of all cyclical paths of a given CFG to transform it into a bipartite Directed Acyclic Graph (DAG). This process effectively transforms the given CFG into a tree that can be passed though the WNTD function. Algorithm 1 presents the general implementation of the described procedure.

4.4.2.3 WNTD Best Match

The evaluation of the best match for a given path against a set of others relies on the underlying paths length and weight. These values are pre-calculated for all paths, stored and reused when needed. The description of how the weights are determined and assigned is further elaborated on

Algorithm 1: WNTD

```
Input: $W[]$: Path weights of function $W$ stored in a linked list.
Input: $BTree_V$: Path weights of function $V$ stored in a B$^+$tree.
Output: $WNTD$: Dissimilarity score between functions $W$ and $V$.
Function WNTD($W$,$BTree_V$)
    $sum \leftarrow 0$ ; $sum_W \leftarrow \sum_{j=1}^{N}(w[j])$;
    $sum_V \leftarrow \sum_{j=1}^{M}(v[j])$;
    foreach $w[i] \in W$ do
        $v_{BM}$ = exactMatch($BTree_V$, $w_i$) ;
        if $v_{BM}$ != -1 then
            $sum+= \left|\frac{w[i]}{sum_W} - \frac{v_{BM}}{sum_V}\right|$; ;
            $W.remove(w[i])$;
        end
    end
    $v_{BM} \leftarrow 0$ ;
    foreach $w[i] \in W$ do
        $v_{BM}$ = inexactMatch($BTree_V, w_i, \delta$); ;
        $sum+= \left|\frac{w[i]}{sum_W} - \frac{v_{BM}}{sum_V}\right|$
    end
    return $WNTD = sum/2$;
end
```

in Subsections 4.4.2.4 and 4.4.2.5. In order to efficiently identify the best match, a B+ tree data structure is utilized.

This data structure is k-ary. This implies that it has a root node, and each of its nodes do not have more than k children. In the case of B+ trees, they often have a large value of k. What distinguishes a B+ tree from a B tree is that only leaf nodes contain values, and each leaf node has pointers to its previous and next sibling leaf node.

This structure is leveraged when comparing paths. Given two paths, all path weights of the largest one are placed inside a B+ tree. The smaller ones weights are pre-sorted in ascending order. Then for each path, the nearest lower and upper bound value are identified in logarithmic time inside the B+ tree. From there, the matches are assessed and the nearest one whose distance is less than the determined threshold γ is kept. If this is the case, the matched value is removed from the B+ tree or else the path is considered not to have found a match. Unmatched paths yield a zero similarity score. Thus, the complexity of this linear iteration over the B+ tree to identify matches has a $O(n \log n)$ complexity.

4.4.2.4 Mnemonic Instructions Grouping

In Assembly, mnemonics are used as a symbolic representation of a machine instruction or operation code. These opcodes are machine-level instructions that indicate what actions should be performed. Thus, the mnemonics that are present in a given function provide information as to its semantics. That is the overall meaning and purpose that a given function has. This can be seen through previous work that leveraged function semantics. It includes the use of opcode frequencies to detect the presence of malicious code [164], and metamorphic viruses [165]. This utilization of opcode frequencies shows that it is possible to leverage them to gain information as to a functions capabilities. For example, there should be a relatively higher concentration of mathematical and logical operations in a hashing function implementation when comparing it to another that writes to files.

In this context, equivalent instruction replacement [165] is an issue. Current day processors can have multiple instructions that perform the same actions. Instruction replacement is thus the process of replacing a given instruction by another equivalent one which can lead to misleading opcode frequency results. In our scenario, this problem manifests itself through the use of different compilers and optimization flags during the compilation process. For example, in certain circumstances the MOV and MVN mnemonics can be substituted for each other [177, 178]. It is thus necessary to regroup and effectively generalize such mnemonics in order to avoid this dilemma. The first step in this process is to identify all mnemonics presented in the ARM architecture [179]. These can then be classified based on their functionality into groups such as comparison instructions and stack instructions.

Once the groups have been created, it is necessary to identify which of them are the most useful and relevant. This is in order to improve the final overall accuracy and efficiency of the CFG basic block weight assignment process (described in Subsection 4.4.2.5). As is performed in Subsection 4.4.1.2, MI is utilized to assess which group combination provides the most information. In order to perform this process, we extracted the frequencies of all mnemonic groups from all functions in our repository. Figure 18 illustrates the received mnemonic grouping results. From them we select the top ones, namely the first seven groups to be further utilized in computing CFG basic block weights.

4.4.2.5 Weight Assignments

In order to properly asses the WNTD of two given CFGs, it is necessary to first determine the weights that are associated to each of their paths. The approach discussed here is based off the concept of graph kernels with linear time complexity [180, 181]. The idea is to create a single representative hash, which can be used as a weight, for each basic block in a given CFG. Then for

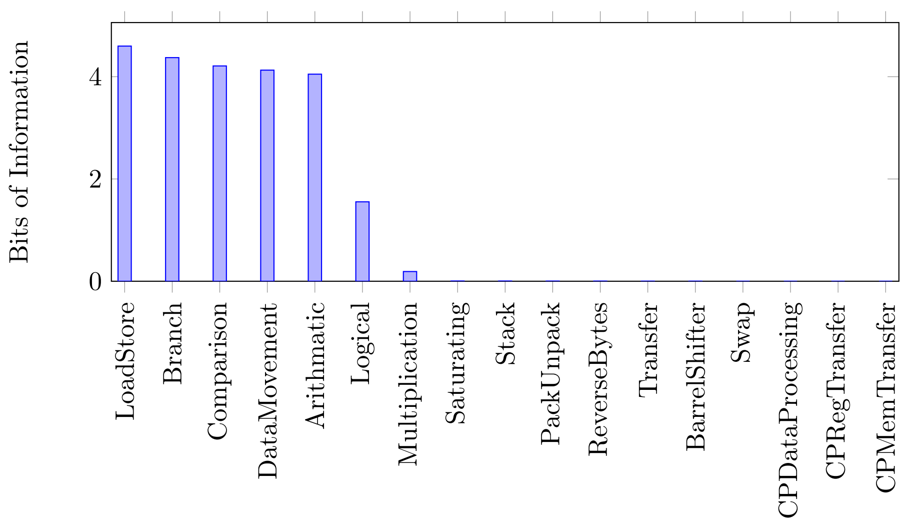


Figure 18: Mutual Information of Mnemonic Groups

a given path in the CFG, each of the weighted nodes on it can be combined in order to create a single path weight. This value can then be used in performing the branch-based detection phase.

The mnemonics from the most informative groups identified in Subsection 4.4.2.4 are leveraged here as the core of a given basic blocks weight. First, a feature vector is created where each index represents the presence of a given instruction. The value stored at this index is the number of times the given instruction appears in the block. This effectively creates a Probability Density Function (PDF) of the instructions that are present in the basic block.

It is however also desired to take into account the relative location that a given basic block has within a given CFG. This is achieved by leveraging its in and out degrees though its parent and child vertices. As such, a basic blocks feature vector is intersected (join) with all of its parents ones and combined (union) with its children. The result of this process is a location sensitive feature vector. However, it still needs to be converted into a single representative value. Furthermore, similar feature vectors need to be converted to similar values with the least collisions as possible. To solve this problem, the LSH [182, 183] algorithm is employed. LSH generates a hash value for a given input, where similar inputs provide similar outputs. As such, a basic blocks feature vector can be reduced to a single value that can be easily compared.

4.4.3 Fuzzy Matching-Based Detection

The final fuzzy matching procedure takes place on all remaining results from the branch-based detection phase. These are a relatively small set of functions that have already been assessed as to having a high degree of similarity to the provided target function. Inspired by [122], each provided function is compared to and ranked by similarity to the target function. The results of this phase are ordered pairs of the highest identified matches.

4.4.3.1 Longest Path Generation

The function matching approach can de decomposed into three steps namely (i) longest path generation, (ii) path exploration and (iii) neighborhood exploration. For a given target function CFG, all of its loops are first unrolled, followed by the use of the Depth First Search (DFS) algorithm to identify its longest path. A path is then chosen as the basis of comparison since it represents the functionality created by the ordered execution of all the basic blocks it contains. Leveraging two equivalent or highly similar execution paths is a good basis for the comparison process. Thus the longer the path, the larger the possibility to acquire matching pairs of basic blocks. Leveraging another default path, such as the shortest, would give lower accuracy results. The reason is that the shortest paths are often used for error checks that bypass most of the functions execution [184].

4.4.3.2 Path Exploration

Given the longest target function path, it is paired with its best match amongst all those the reference function contains. This can be achieved by leveraging the work performed by [122, 185] through combining the Breadth First Search (BFS) algorithm in order to identify all paths and the LCS algorithm [24] to identify the best one. Within the context of the LCS algorithm, each basic block is compared using a similarity function. It leverages the instruction contents of each block as the basis of its comparison operation. Thus, the path from the reference function that contains the most similar blocks to the longest target path is considered to be the best match. This result is determined by applying backtracking to the contents of the memorization table the LCS algorithm produces. All resulting basic block matches are inserted into a priority queue.

Inspired by [186], the algorithm leverages the longest identified target function path along with the entire reference function CFG. A dynamically sized memoization table, that can accommodate the unpredictable size of CFGs, is used to store all comparison results. A vector is employed to store the highest identified similarity score and a queue is leveraged to keep track of all nodes that are to be explored. The algorithm is initialized with the root node of the provided reference function. Then for each node of interest, the LCS with respect to the longest path. Finally if the resulting score is superior to all others, it is preserved and all successor nodes are added to the queue for further investigation.

4.4.3.3 Neighborhood Exploration

The final step in this procedure is the utilization of the Hungarian algorithm on the highest matching pair of paths. This is necessary in order to perform a neighborhood exploration relative to the initial basic block matchings. As such, by leveraging the in and out degrees of the highest matching basic block pairs in the identified paths, it is possible to locate similar neighbors they possess. This has the effect of increasing the overall accuracy of the initial pairing and produces the final output of the fuzzy comparison process. It is composed of the identified set of target and reference function basic block pairs, along with a similarity score for each pair. A final similarity score has to then be computed by leveraging this information. This formula is formally defined in Equation 7 where f_T is a target function with n_T basic blocks, f_r is a reference function with n_r basic blocks, k is the number of matched basic blocks between the target and reference function, and $WJ(S,T)$ returns the similarity score between the two matched basic blocks.

$$similarity\ (f_T, f_r) = \frac{2 \times \sum_{i=1}^{k} WJ(S,T)}{n_T + n_r} \tag{7}$$

4.4.3.4 Basic Block Similarity Comparison

The basic block similarity computation proposed by [122] leverages the LCS algorithm and applies it to the instructional contents of two given basic blocks. Although effective, this approach suffers from two main limitations. The first is instruction reordering, which reduces match accuracy, since the LCS algorithm is instruction order sensitive. The second are instruction substitutions that are syntactically different but semantically equivalent, which will also affect similarity scores. Due to these limitations, we chose to follow a different approach that can mitigate them. The Weighted Jaccard (WJ) similarity [187] has been selected as the means of comparing the instructions of two basic blocks. It is necessary to include weights, in the form of instruction frequencies, in order to prevent possible false positives. Purely based on the nature of the formula, instruction reordering does not affect the final similarity score. Furthermore, it is possible to handle instruction substitutions by grouping them as discussed in Subsection 4.4.2.4. Another advantage is that this comparison method can be implemented to run in linear time. The utilized WJ formula is defined in Equation 8 where S and T are sets of basic block mnemonic frequencies, as well as n and m are the number of elements in each block.

$$WJ(S,T) = \frac{\sum_{k=1}^{N} \min(S_k \cap T_k)}{\sum_{k=1}^{N} \max(S_k \cup T_k)},\ N = \max\{m, n\} \tag{8}$$

4.5 Framework Design and Implementation

During the entire course of this work, we design and implement a framework that is able to perform the aforementioned capabilities. This entails specific system requirements that define what a user can do with it. It is thus necessary to create specific user stories that defined what general capabilities it needs to perform. It is then necessary to perform a thorough design process where the general topology, structure and used technologies have to be determined. Once completed, this information is used to perform a proper implementation.

4.5.1 User Stories

A user story is a simple description of a feature viewed by a user's perspective. It allows for the creation of the description of a system requirement that is to be implemented. An epic describes a complex feature that is broken down into several smaller user stories. This framework is meant to be a private in-house implementation. The main individuals who will interact with it are the members of the laboratory. As such, three main epics have been identified: uploading a target binary, uploading a reference binary and performing a comparison. These along with a set of general user stories, have been completed in order for the framework to be fully implemented and considered functional. All defined user stories are presented in Tables 17 through 22.

Table 17: Epic: Upload a Target Binary

Id	As a	I want to	so that I can
1	user	select a binary from my file system	upload it.
2	user	input the binaries name	easily identify it in the future.
3	user	input the binaries version	easily identify it in the future.
4	user	select an IDC file from my file system	import my custom disassembly into IDA Pro.

Table 18: Epic: Manage Target Binaries

Id	As a	I want to	so that I can
5	user	view all uploaded target binaries	get an idea of the state of the system.
6	user	view all uploaded target functions	get an idea of the state of the system.
7	user	search through uploaded targets	easily select those I want to compare.
8	user	delete a target binary	remove no longer needed binaries.
9	user	delete a target function	remove no longer needed functions.

Table 19: Epic: Upload a Reference Binary

Id	As a	I want to	so that I can
10	user	select a binary from my file system	upload it.
11	user	input the binaries name	easily identify it in the future.
12	user	input the binaries version	easily identify it in the future.

Table 20: Epic: Manage Reference Binaries

Id	As a	I want to	so that I can
13	user	view all reference binaries	get an idea of the state of the system.
14	user	view all reference functions	get an idea of the state of the system.
15	user	search through references	easily select those I want to compare.
16	user	delete a reference binary	remove no longer needed binaries.
17	user	delete a reference function	remove no longer needed functions.
18	user	mark reference functions as vulnerable	easily identify potential vulnerabilites.

Table 21: Epic: Perform a Comparison

Id	As a	I want to	so that I can
19	user	select target functions	use them to perform a comparison.
20	user	select reference functions	use them to perform a comparison.
21	user	examine matching CFGs	determine if they have any vulnerabilities.
22	user	view the results of the comparison	see all of the matching results.
23	user	view the comparison result between a specific target and reference function	possibly confirm the presence of a vulnerability.

Table 22: General User Stories

Id	As a	I want to	so that I can
24	user	see the status of the application	easily see if it is running.
25	user	see the configuration of the application	see how it will behave in real-time.
26	user	edit its configuration in real-time	adapt its behavior to my needs.
27	user	cache certain functions in memory	perform faster comparisons.

4.5.2 Design

We decide to create a web-based application as opposed to a desktop one. Such an approach provides several advantages. Any individual in the lab can access the framework without having to set it up locally in their own machine. All they need is access to a standard web browser to be able to leverage it. This saves them both time and effort. It also allows us to deploy the application on a powerful server so that it performs better and not be limited by storage space.

We decide to use a classic three-tier client server based architecture that exposes a JSON based REST API that can be consumed by other services. The key constraints that a REST service has are clear advantages in this scenario. REST establishes that its service must follow a client server architecture, which is the case here. There are no state concerns in this scenario. Thus, statelessness is a benefit due to reduced overhead. Caching is recommended in such scenarios. It is a very good design choice in this situation in order to store a limited number of comparison results and database entries. The uniform interface principle of REST enables BinArm to be easily interoperable with other services. This is the case for its web interface which has been previously created for BinSequence [122]. This allows for a large amount of code re-use between the two projects and greatly accelerated this one's development. It also enables the creation of benchmarking scripts that could directly interact with it.

All received and analyzed binary data results need to be stored for future use. In order to do so, a specific database technology is utilized. Due to the large volume of data that is stored in this database, it is recommended to use a NoSQL database since they are made for high-performance processing of large datasets at a scale. Thus, the obvious contenders for use are Apache Cassandra [188], Couchbase [189], HBase [190], and MongoDB [191]. In this scenario, a fast read time is the most important factor. This is because the comparison process, which is the main purpose of the framework, only performs read operations. It is only when uploading a new binary, which is far less often, that a database write occurs. According to a benchmark of top NoSQL databases performed by End Point [192], Apache Cassandra has the best performance for a read-mostly workload and is used.

Deployment of a project can be a very difficult process. Generally it is needed to re-create a specific system setup that can run a given program. This issue can be completely avoided with the use of Docker [193]. This tool allows the creation of a standalone software package that can be easily deployed. Thus, a single stable environment inside a docker container has to be created once and can be re-deployed just about anywhere. Such an approach saves both time and effort. Figure 19 provides an overview of the frameworks deployment structure.

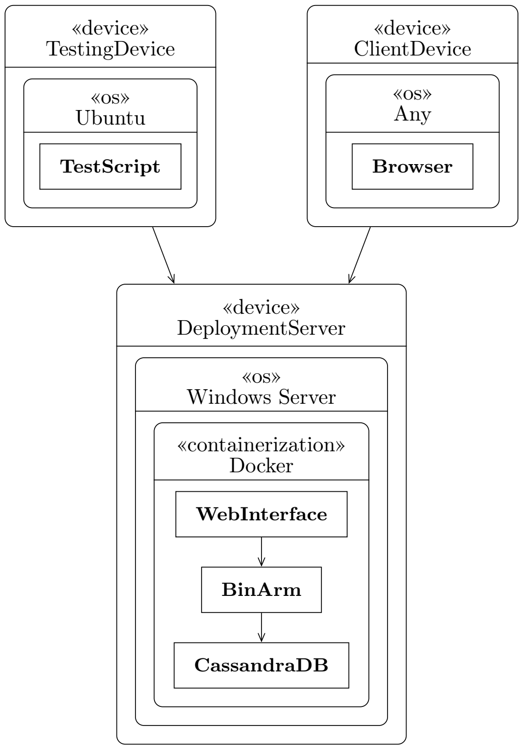


Figure 19: Deployment Diagram

It is then possible to move onto the formal design of the system. In order to do so, domain models are utilized. These are conceptual diagrams that describe all the data and components of a system, along with their relationships and interactions. All the domain model components are grouped into several logical packages. Each of these represents a specific functionality group of the application. The first model of interest is the general one that describes the entry-point of the program along with all API handlers. The entry point is responsible to initialize the application. A router is responsible to handle all incoming requests and redirect them to their appropriate handler. This structure is illustrated in Figure 20.

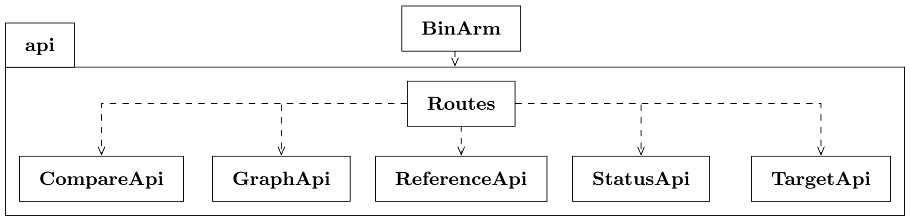


Figure 20: Entry-point Domain Model

The different REST interface handles are divided by their respective user stories and functionality. The TargetAPI model in Figure 21 illustrates the general components needed to implement all stories in the Upload a Target Binary (Table 17) and the Manage Target Binaries (Table 18) epics. The TargetFunctionCache is a Least Recently Used (LRU) cache that stores all recently accessed target functions. There is a model used to represent a target function. There are a few helper models used to facilitate the creation of responses.

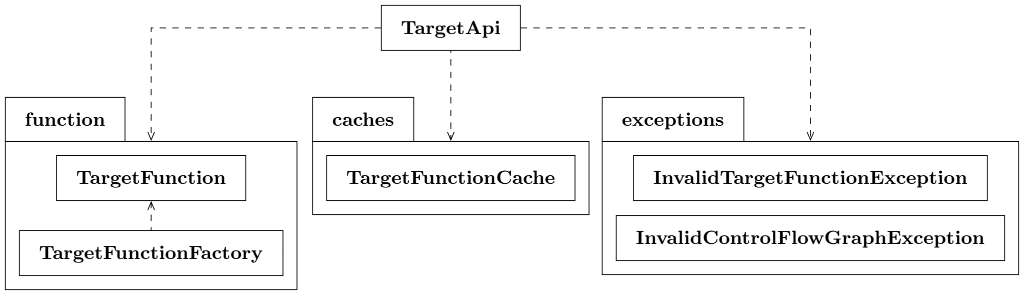


Figure 21: Target Domain Model

The ReferenceApi model in Figure 22 illustrates the general components needed to implement all stories in the Upload a Reference Binary (Table 19) and the Manage Reference Binaries (Table 18) epics. The ReferenceFunctionCache is a LRU cache that stores all recently accessed reference functions. There is a model used to represent a reference function. There are a few helper models used to facilitate the creation of responses.

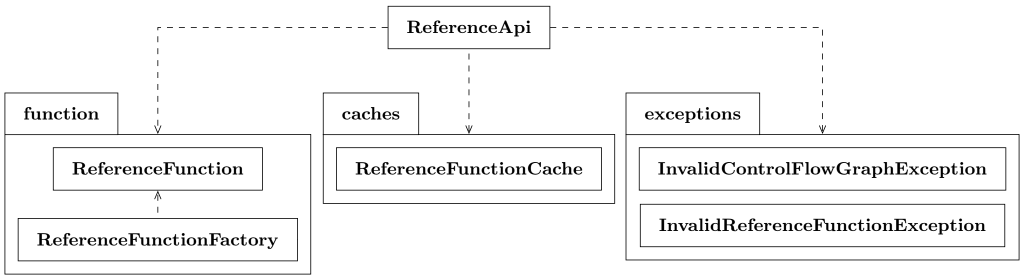


Figure 22: Reference Domain Model

The ComparisonApi and GraphApi models in Figures 23 and 24 illustrate the general components needed to implemented all stories in the Perform a Comparison epic (Table 21). They use a ComparisonCache, which is a LRU cache that stores all recently computed ComparisonResults. The FiltrationHelper is used to perform the multi-staged comparison approach.

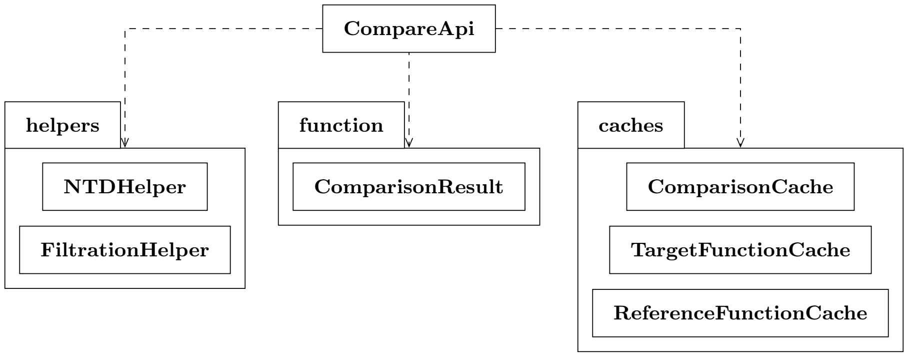


Figure 23: Comparison Domain Model

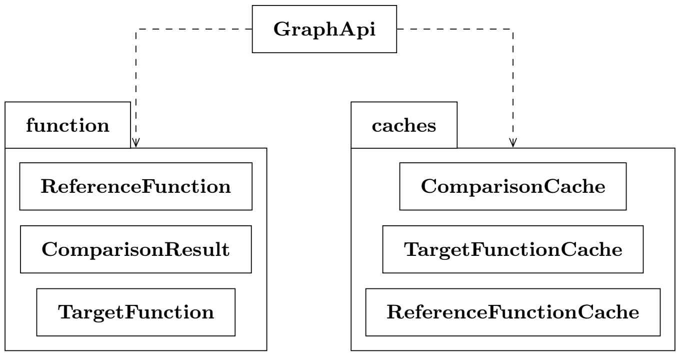


Figure 24: Graph Domain Model

The LoadApi, the SettingsApi, and the StatusApi models in Figures 25, 26, and 27 illustrate the general components needed to implement all general user stories (Table 22).

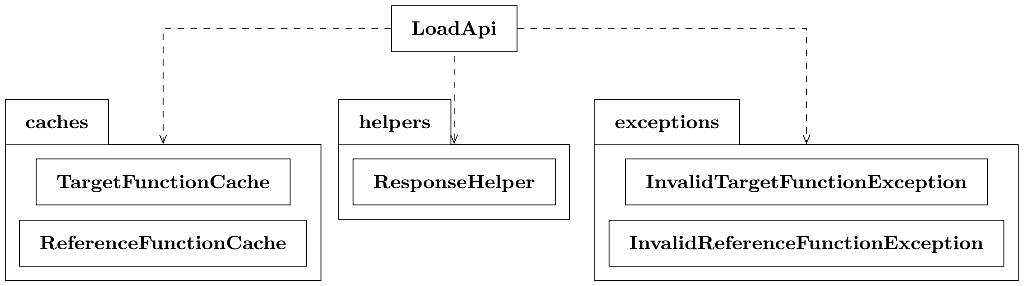


Figure 25: Load Domain Model

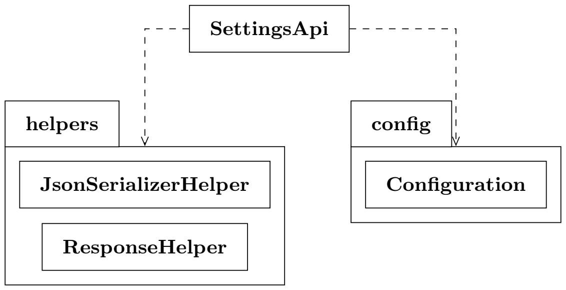


Figure 26: Settings Domain Model

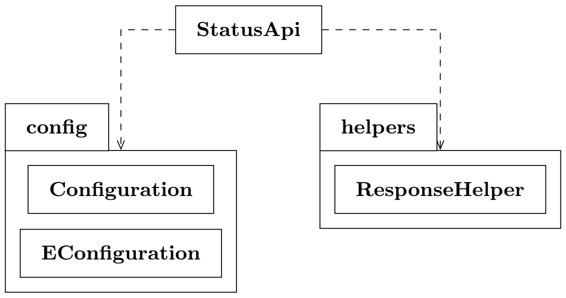


Figure 27: Status Domain Model

The remaining domain models in Figures 28, 29, and 30 provide some general overview of other miscellaneous components that are required to complete all user stories. The CFG domain model shows how functions are represented in memory, along with the database model that enables their storage and retrieval. The configuration model is utilized by most others to provide critical behavioral information.

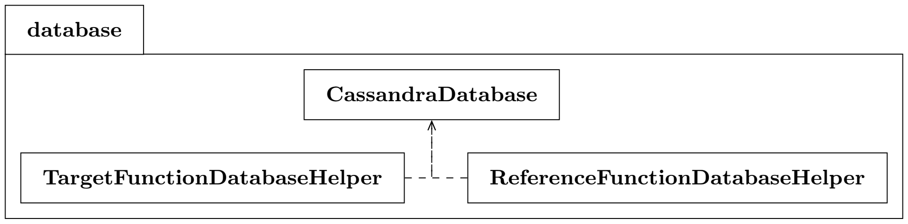


Figure 28: Database Domain Model

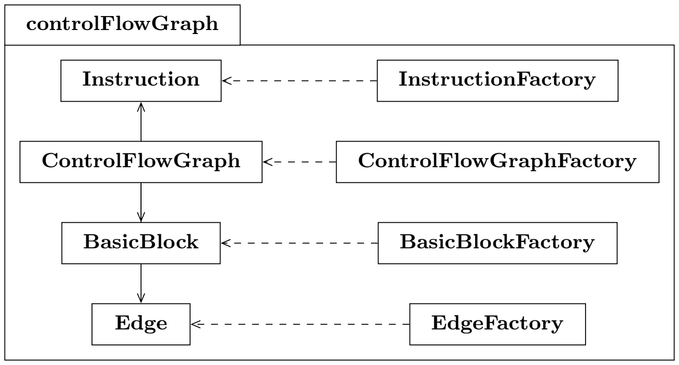


Figure 29: Control Flow Graph Domain Model

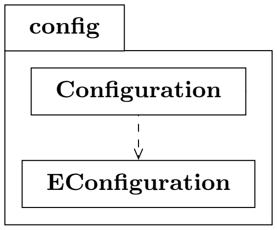


Figure 30: Configuration Domain Model

4.5.3 Technology Stack

There are several technologies utilized in the implementation process. This section aims to provide a general overview of them along with justification as to why they are the most suitable choices.

4.5.3.1 Language

There are two main requirements for the use of the primary programming language for this project; speed and library availability. Speed is key because the efficiency of the framework is used as a benchmark of its capabilities. Thus, it is not wise to use a language such as Python [194]. Libraries are critical to increase implementation time and reduce possible implementation errors or bugs. For such a requirement, it is not wise to use a language such as C, which does not provide an easily usable graph library such as Boost C++. Thus, C++ is chosen since it meets the above-stated requirements.

4.5.3.2 Libraries

Several libraries are utilized to implement BinArm. A core component is the REST framework that is used to implement all HTTP requests. Pistache [195] is an elegant and light weight framework that is extremely easy and intuitive to use. When compared to other alternatives such as cpprestsdk [196] and restbed [197] it outperforms them [198]. JsonCpp [199] is used for all JSON processing and manipulation. It is a core part of the web service for it is used to handle all incoming and outgoing request bodies. Boost [200] is an extremely far reaching and powerful library for C++. Several of its sub-libraries such as multiprecision [201] and graph [202] are utilized for graph operations and handling of large floating point numbers. Logog [203] is a logging library that is specifically made for performance-oriented programs. This makes it very suitable for this work. Trendmicro TLSH [183] is the locality-sensitive hashing library that is used to convert CFG weights into a single hash. What is distinctive with this type of hash is that similar inputs do yield similar outputs. DataStax Cpp Driver [204] is the utilized library that allows the application to interact with its database. The utilized LRU cache implementation is a pure C++11 header only implementation that is taken from Github [205].

4.5.3.3 Other Technologies

There are a couple of other utilized tools that are worthy of note. Docker Compose [206] is leveraged to easily define and orchestrate the deployment of all the application containers. This is done using a YAML file in which all configuration can be defined. IDA Pro [86] is a very powerful dissembler that is used to extract all CFG data from given binaries before it is fed into BinArm. Python [207] is used to create all BinArm evaluation scripts and to perform all function shape-based, and branch-based detection clustering and feature selection computations. Vagrant [208] is used to create a virtual machine environment that is specifically designed to perform reverse engineering and firmware analysis.

4.5.4 Usage Scenario

Here, we provide a quick examination of how the framework is employed through a simple firmware vulnerability analysis scenario. Figure 31 provides a general overview of the default interface. It displays a wide range of information such as the indexed targets, the vulnerability database contents and the total number of indexed functions.

4.5.4.1 Firmware Indexing

We acquire a copy of the NI PMU 1.0.11 firmware image, unpack it and extract the binaries that are within its file system. It is desired to analyze its copy of libcrypto and as such it is uploaded as can be seen in Figure 32. This is done by clicking on the Upload a binary button on the target half of the interface. From there the library and its information can be provided to BinArm.

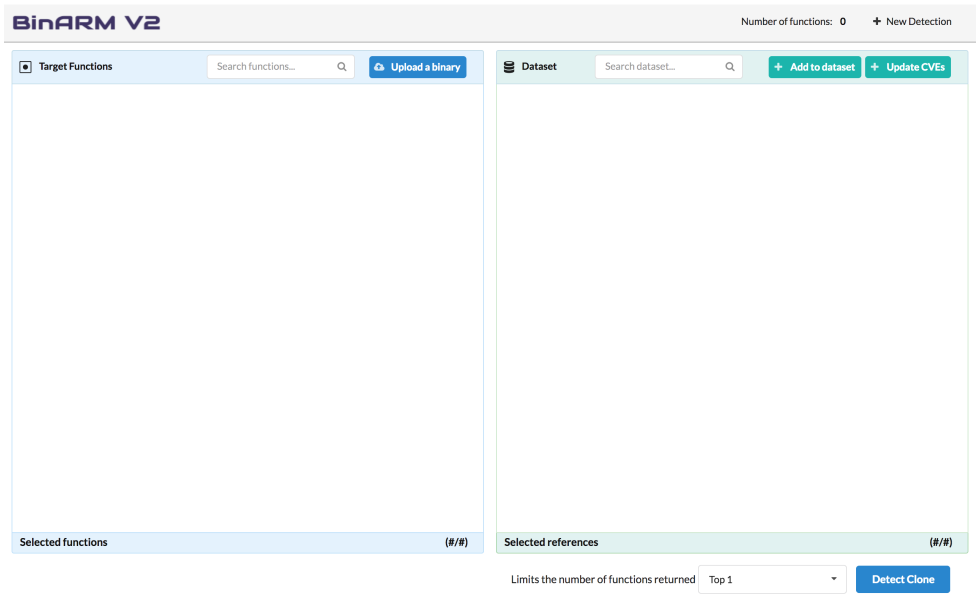
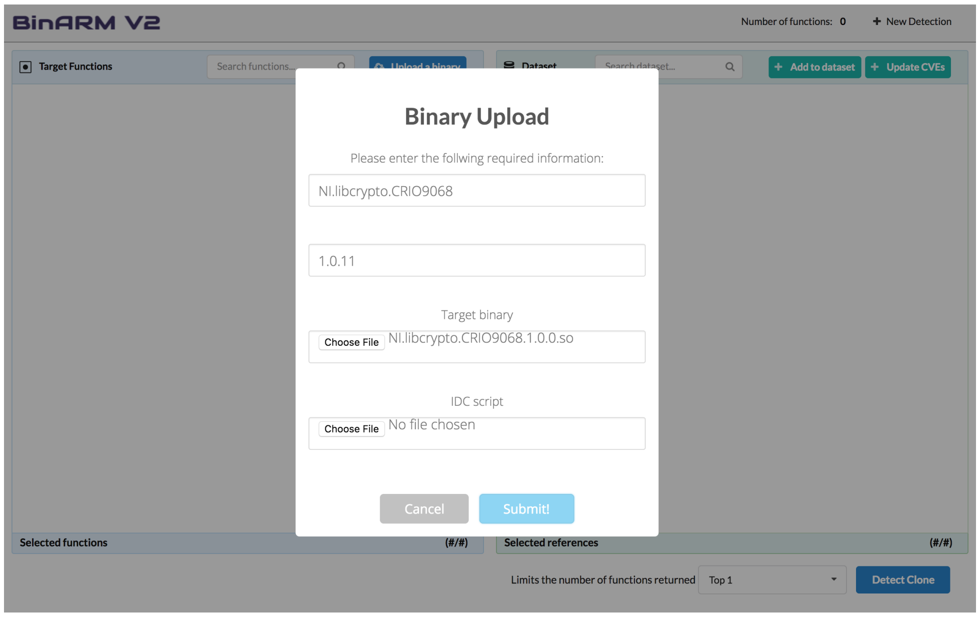


Figure 31: Default Interface



Binary Upload

Please enter the follwing required information:

NI.libcrypto.CRIO9068

1.0.11

Target binary

Choose File NI.libcrypto.CRIO9068.1.0.0.so

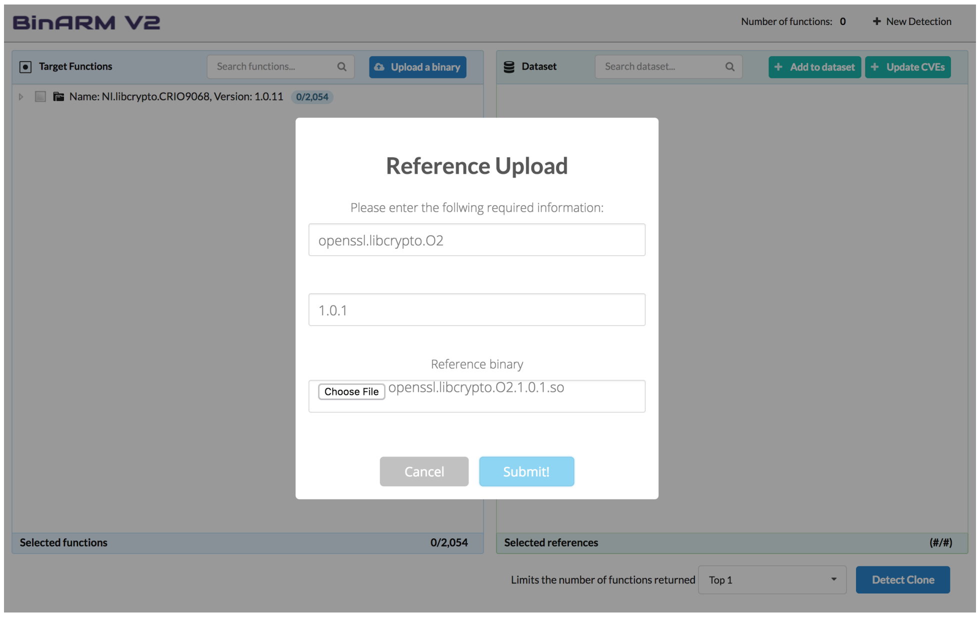
IDC script

Choose File No file chosen

Figure 32: Firmware Indexing

4.5.4.2 Vulnerability Indexing

The same procedure is then repeated for the dataset library that contains the evaluated vulnerabilities. We acquire and compile libcrypto version 1.0.1 with the O2 flag and index it as can be seen in Figure 33. This is similarly done by clicking on the Add to dataset button on the dataset half of the interface.



Reference Upload

Please enter the follwing required information:

openssl.libcrypto.O2

1.0.1

Reference binary

openssl.libcrypto.O2.1.0.1.so

Figure 33: Vulnerability Interface

It is then required to indicate which functions within the uploaded library are vulnerable. This is done through the use of a structured JSON file, a sample of which can be seen in Figure 34. This file can be provided by clicking the Update CVEs button. This process results in each of the identified vulnerable functions being labeled with their respective CVEs.

4.5.4.3 Firmware Evaluation

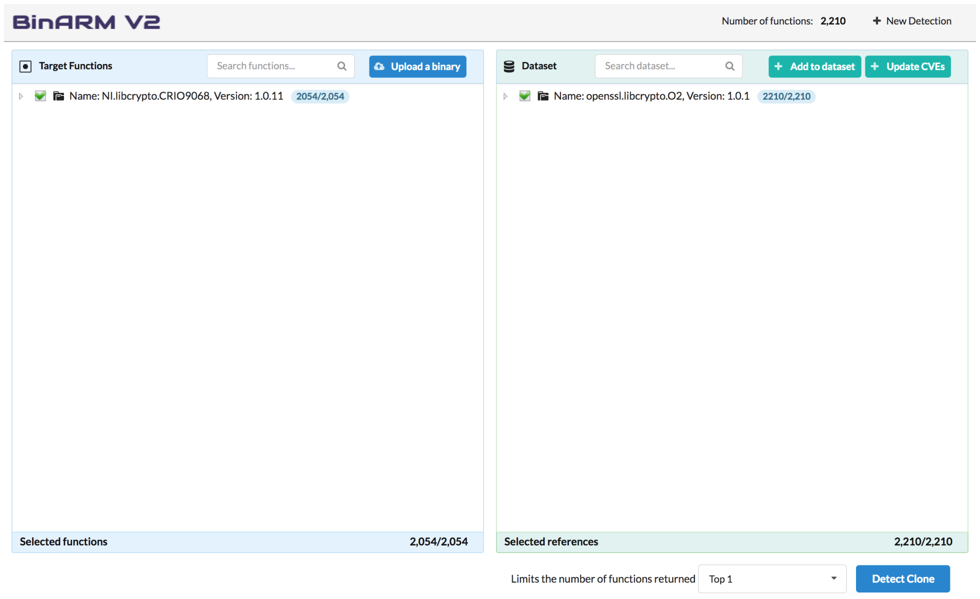
The two uploaded binaries can then be compared by selecting them and clicking the Detect Clone button as seen in Figure 35. It is also possible to select the top number of matches for each compared firmware function that should be returned.

Figure 36 gives a general overview of the results returned from an analysis. Each matching pair is presented along with its score and potential CVE. It is possible to sort each column to view the results in any desired order. If a given match is of particular interest, it is possible to inspect it

```
{
  "Libraries": [
    {
      "Functions": [
        {
          "Function": "tls1_process_heartbeat",
          "CVE": "CVE-2014-0160"
        },
        {
          "Function": "dtls1_process_heartbeat",
          "CVE": "CVE-2014-0160"
        }
      ],
      "Version": "1.0.1",
      "Library": "openssl"
    }
  ]
}
```

Figure 34: Vulnerability Marking

further by simply clicking on it. This will provide the CFG comparison overview that can be seen in Figure 37. It allows an analyst to view all basic block pairs along with their similarity score. Code differences can be further inspected to assist in validating the accuracy of the match.



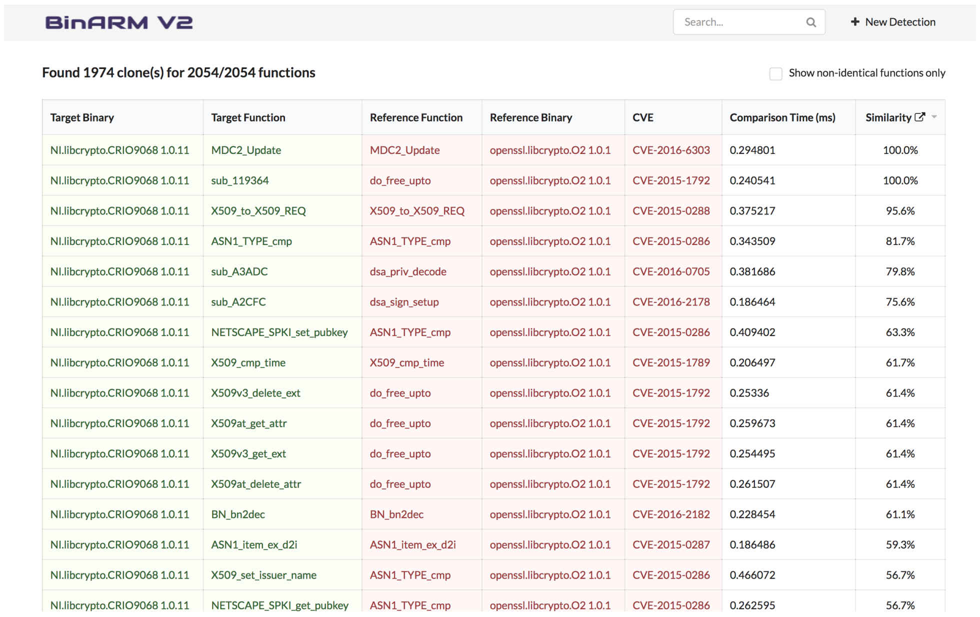
Name: NI.libcrypto.CRIO9068, Version: 1.0.11 2054/2,054

Name: openssl.libcrypto.O2, Version: 1.0.1 2210/2,210

Selected functions 2,054/2,054

Selected references 2,210/2,210

Figure 35: Firmware Evaluation



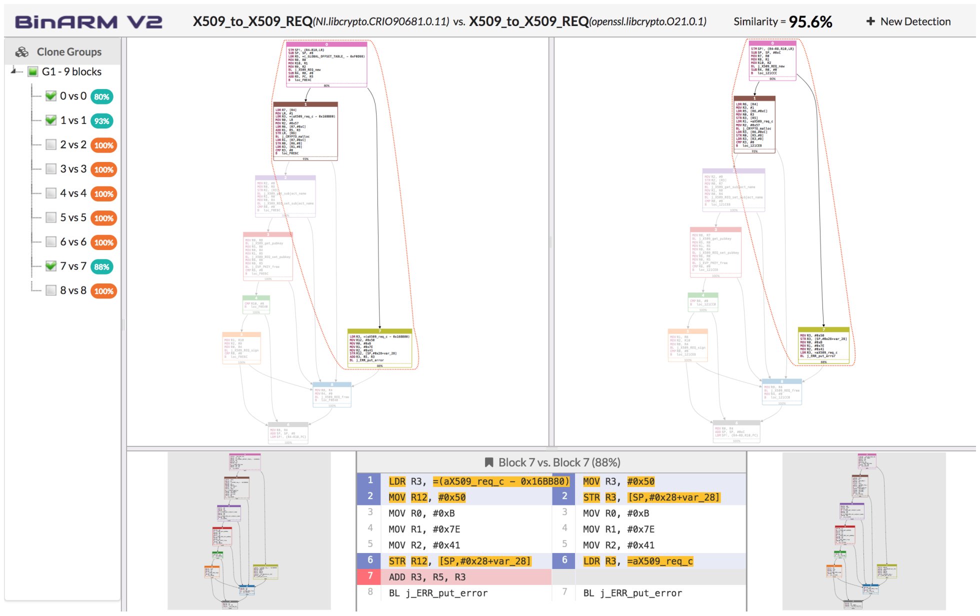
BinARM V2

Found 1974 clone(s) for 2054/2054 functions

Show non-identical functions only

Target Binary	Target Function	Reference Function	Reference Binary	CVE	Comparison Time (ms)	Similarity
NI.libcrypto.CRIO9068 1.0.11	MDC2_Update	MDC2_Update	openssl.libcrypto.O2 1.0.1	CVE-2016-6303	0.294801	100.0%
NI.libcrypto.CRIO9068 1.0.11	sub_119364	do_free_upto	openssl.libcrypto.O2 1.0.1	CVE-2015-1792	0.240541	100.0%
NI.libcrypto.CRIO9068 1.0.11	X509_to_X509_REQ	X509_to_X509_REQ	openssl.libcrypto.O2 1.0.1	CVE-2015-0288	0.375217	95.6%
NI.libcrypto.CRIO9068 1.0.11	ASN1_TYPE_cmp	ASN1_TYPE_cmp	openssl.libcrypto.O2 1.0.1	CVE-2015-0286	0.343509	81.7%
NI.libcrypto.CRIO9068 1.0.11	sub_A3ADC	dsa_priv_decode	openssl.libcrypto.O2 1.0.1	CVE-2016-0705	0.381686	79.8%
NI.libcrypto.CRIO9068 1.0.11	sub_A2CFC	dsa_sign_setup	openssl.libcrypto.O2 1.0.1	CVE-2016-2178	0.186464	75.6%
NI.libcrypto.CRIO9068 1.0.11	NETSCAPE_SPKI_set_pubkey	ASN1_TYPE_cmp	openssl.libcrypto.O2 1.0.1	CVE-2015-0286	0.409402	63.3%
NI.libcrypto.CRIO9068 1.0.11	X509_cmp_time	X509_cmp_time	openssl.libcrypto.O2 1.0.1	CVE-2015-1789	0.206497	61.7%
NI.libcrypto.CRIO9068 1.0.11	X509v3_delete_ext	do_free_upto	openssl.libcrypto.O2 1.0.1	CVE-2015-1792	0.25336	61.4%
NI.libcrypto.CRIO9068 1.0.11	X509at_get_attr	do_free_upto	openssl.libcrypto.O2 1.0.1	CVE-2015-1792	0.259673	61.4%
NI.libcrypto.CRIO9068 1.0.11	X509v3_get_ext	do_free_upto	openssl.libcrypto.O2 1.0.1	CVE-2015-1792	0.254495	61.4%
NI.libcrypto.CRIO9068 1.0.11	X509at_delete_attr	do_free_upto	openssl.libcrypto.O2 1.0.1	CVE-2015-1792	0.261507	61.4%
NI.libcrypto.CRIO9068 1.0.11	BN_bn2dec	BN_bn2dec	openssl.libcrypto.O2 1.0.1	CVE-2016-2182	0.228454	61.1%
NI.libcrypto.CRIO9068 1.0.11	ASN1_item_ex_d2i	ASN1_item_ex_d2i	openssl.libcrypto.O2 1.0.1	CVE-2015-0287	0.186486	59.3%
NI.libcrypto.CRIO9068 1.0.11	X509_set_issuer_name	ASN1_TYPE_cmp	openssl.libcrypto.O2 1.0.1	CVE-2015-0286	0.466072	56.7%
NI.libcrypto.CRIO9068 1.0.11	NETSCAPE_SPKI_get_pubkey	ASN1_TYPE_cmp	openssl.libcrypto.O2 1.0.1	CVE-2015-0286	0.262595	56.7%

Figure 36: Evaluation Results



BinARM V2 X509_to_X509_REQ(NI.libcrypto.CRIO90681.0.11) vs. X509_to_X509_REQ(openssl.libcrypto.O21.0.1) Similarity = 95.6%

Block 7 vs. Block 7 (88%)

```
1 LDR R3, =(aX509_req_c - 0x16BB80)   1 MOV R3, #0x50
2 MOV R12, #0x50                      2 STR R3, [SP,#0x28+var_28]
3 MOV R0, #0xB                        3 MOV R0, #0xB
4 MOV R1, #0x7E                       4 MOV R1, #0x7E
5 MOV R2, #0x41                       5 MOV R2, #0x41
6 STR R12, [SP,#0x28+var_28]          6 LDR R3, =aX509_req_c
7 ADD R3, R5, R3
8 BL j_ERR_put_error                  7 BL j_ERR_put_error
```

Figure 37: Result Comparison

4.6 Summary

The purpose of our proposed framework dubbed BinArm [150] is to perform a fuzzy known vulnerability matching in IED firmware. This general process can be subdivided into two main steps namely (i) *offline preparation* and (ii) *online search*.

In the offline preparation phase, two large function repositories are created. For both it is essential to identify a set of prominent smart grid manufacturers. This information is first leveraged in order to identify and acquire a multitude of relevant and publicly available firmware images that they have produced. These are further filtered down by relevance criteria such as CPU architecture. The firmware database is then populated with these acquired, disassembled and processed IED firmware images. The list of smart grid manufacturers is then further leveraged in order to identify sources of utilized open-source software in their released firmware images. This software reuse information can now be combined with known vulnerabilities taken from the CVE [26] database. This allows for the creation of a list of most commonly used vulnerable libraries within the industry. This information can be further expanded into a list of most likely present vulnerable functions.

The online search phase performs the core functionality in BinArm. It performs the actual fuzzy vulnerability identification procedure by leveraging the two previously mentioned sources of information. In order to make this process as efficient as possible, this phase performs a multi-stage detection process that is composed of three main fuzzy matching phases. These each aim at gradually identifying the best candidate matches for a given function while discarding those that are not. Each phase is gradually more thorough and complex. This is to quickly and cheaply, in terms of computation time, identify a limited set of possible candidates that can be more carefully examined. The three stages are represented by logical components which are: (i) shape-based detection, (ii) branch-based detection, and (iii) fuzzy matching-based detection . Shape-based detection leverages the concept of a functions shape, which is defined as the set of heterogeneous features that can be extracted from it [22]. By leveraging this work, it is possible to identify a concrete list of usable features. Thus, this phase leverages the most important and relevant of these features in order to assist in pruning the initial search space. This is done by first evaluating our entire general database with respect to the identified features. It is then possible to identify which of them are the most relevant in our given context. From there, only the most important can be used to calculate a relevance threshold that will determine if a given function should be further analyzed. Branch-Based Detection further reduces the size of a given comparison set. This is in order to increase the overall performance of the function similarity assessment process. It is used to compare functions from a different informational standpoint from the shape-based phase. This is achieved by focussing on function execution paths. The branch-based detection process takes as inputs the CFGs of a given target and reference function. From there, it leverages a weighted path comparison algorithm to assess the distance between the two graphs. The final fuzzy matching procedure takes place on all remaining results from the first two detection phase. This is a relatively small set of functions that have already been assessed as to having a high degree of similarity to the provided target function. Inspired by [122], each provided function is compared to and ranked by similarity to the target function. The results of this phase are ordered pairs of the highest identified matches.

Finally a technical overview of how we design and implement the BinARm framework is given. This entails specific system requirements that define what a user can do with it. It is thus necessary to create specific user stories that defined what general capabilities it needs to perform. It is then necessary to perform a thorough design process where the general topology, structure and used technologies have to be determined. Once completed, this information is used to perform a proper implementation.

Chapter 5

Evaluation

In order to validate the proposed approach and implementation, it is necessary to perform a set of experimental evaluations and analysis of acquired results. With this in mind, it is first necessary to acquire substantial IED firmware and vulnerability datasets. These can then be leveraged within a set of evaluations whose purpose is to validate several facets of BinArm such as its function identification accuracy, efficiency and scalability.

5.1 Datasets

A core aspect of this work is the compilation of three substantial datasets. The first dataset is composed of a large set of reference vulnerabilities. These are specific functions that have been acquired due to their relation with the desired smart grid scope of the work and the presence of one or more known vulnerabilities. The second is to provide a set of target IED related firmware. These are analyzed by leveraging the first dataset in order to determine the presence or absence of known security vulnerabilities. The third is composed of general-purpose binary functions. This corpus of functions is leveraged when performing all scalability related evaluations. Together, these datasets can be used to evaluate and benchmark BinArm. The following contents of this section discusses how the contents of each of these datasets has been identified, assembled and analyzed.

5.1.1 Manufacturer Identification

Due to the scope of this work, it is not possible to utilize any known and available firmware or vulnerability. Each identified piece of code must have a concrete relation to the smart grid world. An ad-hoc approach where a generic scraper gathers all available data is thus not applicable in this context. The main components of a power grid are generation, transmission, substations, distribution and consumption [31]. For the purpose of this work, the overall smart grid context has been reduced to the scope of substations and the networking equipment that allows them to operate. Substations in the smart grid contain a plethora of smart devices in the form of IEDs. This critical infrastructure heavily relies on its underlying communication infrastructures to operate. These communication deployments contain several devices of interest such as routers.

Table 23: Identified Relevant Smart Grid Manufacturers

Manufacturers	Relevant Components				
	Automation Hardware	Smart Meters	Automation Software	Communication	Demand Response
ABB Schweiz AG	●				
Aclara		●			
Cisco			●	●	
Electro Industries (EI)		●			
Elster		●	●	●	
General Electric (GE)		●			
Honeywell					●
Itron		●		●	
Landis+Gear (L+G)		●	●	●	
National Instruments (NI)	●				
Schneider Electric (SE)	●				
Schweitzer Engineering Laboratories (SEL)	●				
Sensus		●		●	
Siemens	●				

With this range of devices in mind, it is possible to identify a set of recognized industry manufacturers that are relevant to these three categories. Fortunately, there is some previous work that looks into the categorization of the smart grid vendor ecosystem. GTM Research has published the results of two reports [209, 210], which indicate by market sector which vendors are most prominent.

The Cleantech Group has also published a smart grid vendor report [211] discussing top U.S. smart grid vendors and market dynamics. This information provides the necessary insight to identify top smart grid manufacturers in the chosen scope. This critical information is further utilized to determine relevant vulnerabilities and firmware for the datasets. Table 23 lists the set of identified manufacturers.

5.1.2 Open-Source Usage Identification

Many of the listed companies rely on the usage of open-source software for their product implementations. This generally entails the legal obligation of publishing documents containing the licenses of all utilized open-source software. By investigating several sources of information pertaining to these manufacturers, it is possible to extract large amounts of open-source usage declarations.

Table 24: Identified Relevant Open-Source Libraries

Library	Manufacturers					
	ABB	Cisco	GE	Honeywell	SE	Siemens
ncurses	●	●	●	●	●	●
openssl	●	●	●	●	●	●
utillinux	●	●	●	●	●	●
pcre	●	●	●	●		●
gzip	●	●	●		●	●
libxml2	●	●	●	●		●
libpcap	●	●	●	●		●
linux	●	●		●	●	●
sed		●	●	●		●
zlib	●	●		●	●	●
uboot	●	●		●	●	●
udev	●	●	●	●		●
expat	●	●		●	●	●
libcurl	●	●		●	●	●
openssh	●	●	●	●		●
iptables	●	●	●	●		●
coreutils		●	●	●	●	●
busybox	●	●		●	●	●
ntp		●	●	●	●	●
netsnmp		●	●		●	●
ppp		●	●	●		●
findutils		●	●	●		●
db		●	●	●		●
dhcp		●	●		●	●
e2fsprogs		●		●	●	●

Sources of information include corporate websites, download centers, FTP search tools and product documentation. This information can then be compiled and filtered for relevance to the current context. On top of the corporate usage it is necessary to explore well-known libraries and tools that are pertinent to the current smart grid scope. This includes libraries for well-known smart grid protocols such as distributed network protocol (DNP3) and sample values (SV). Unfortunately, only Simple Network Management Protocol (SNMP) [212] and Network Time Protocol (NTP) [213] have open-source implementations. Pertinent information that is to be identified through this process includes library names, versions and the names of companies that use them.

There are a few interesting observations that can be made on this information acquisition process. Most acquired information is directly available without any need for authentication. For SEL, GE, and Honeywell, some documents require a basic website account that can be easily created. Siemens website is inconsistent when it comes to authentication; they sometimes require and account for open-source declarations depending on the product. GE is the only manufacturer that has a dedicated page informing the customer to contact them for for their declarations. As for Electro Industries, Landis+Gyr, and Aclara they seem to simply not post their declarations online.

This process yields a total of 120 open-source declarations for all identified manufacturers. When further examined, a total of 2, 125 open-source libraries or products are identified. Table 24 lists the top 25 identified libraries. They are ordered by their respective manufacturer usage and thus prevalence in the industry.

5.1.3 Dataset I - Vulnerabilities

These identified libraries are to be used as vulnerability sources. As such, the number of publicly known CVEs for each of these must be identified. A containerized version of the CVE database [214] has been created to assist with this task. Using Python scripts, it is possible to interact with a library and automatically extract the number of vulnerabilities for each of them. The result of this process yields in the identification of a total of 341 vulnerable libraries or projects that contain a total of 12, 267 vulnerabilities. These can be further classified as 4, 344 kernel-level, 5, 581 application-level and 2, 336 open-source vulnerabilities. These are then all ordered by weighing their importance. It is determined by multiplying the number of CVEs for a given library times the number of companies that utilize it. Table 25 lists the top 25 relevant, vulnerable and used open-source libraries.

Once the set of appropriate libraries are identified, it is necessary to acquire all of their CVE information in a concise and usable format. They must also be filtered by relevance, accessibility, cross compatibility for the ARM architecture and number of usable CVEs. A CVE is considered usable if it is limited to a single functions scope and can be extracted from the compiled library binary. The previously mentioned CVE database is reused for this task. A Python program that interacts with the database and then performs a map reduce process on the results has thus been created. It first executes a query for a given library, filters to ensure that the vulnerability is present at a function level and that the vulnerable version of the library is given and then outputs all results in HTML and CSV formats.

Table 25: Identified Relevant and Vulnerable Open-Source Libraries

Library	Compiled	Number of CVEs
php		601
imagemagick		402
openssl	●	189
mysql	●	564
tcpdump		162
openssh	●	87
ntp	●	79
libtiff		149
postgresql	●	98
ffmpeg		274
pcre		49
python		81
glibc	●	81
qemu		225
libxml2	●	44
bind		102
binutils		97
libcurl	●	34
freetype		83
libpng		47
samba		124
utillinux		15
cups		88
lighttpd	●	28
netsnmp	●	21

Due to time constraints, only a subset of the identified libraries were selected for actual compilation and vulnerability extraction. As such, 10 highly ranked, relatively easy to compile, and C based libraries were selected. The source code for the identified versions of the given libraries are then acquired and cross compiled using all optimization flags. All relevant archive and shared object files are then retrieved. This process results in a total of $44,416$ binaries. These can then be loaded into IDA Pro and disassembled. By using our custom Python script, we are able to extract all functions and their CFGs in the desired format. However, in order to reduce the number of false positives, we ensure that each function has a minimum of 5 basic blocks [120]. From there, all vulnerable functions are identified and stored inside BinArm's vulnerability database for future use. As such, this database consists of a total of $5,103$ vulnerabilities. This number can however be reduced to 235 unique vulnerabilities after discarding the duplicates that are created due to the use of different compilers and optimization flags.

5.1.4 Dataset II - Firmware Images

The next step is to assemble a set of firmware that can be used to assess BinArm and our gathered vulnerability set. These firmware images must preferably be publicly available for ease of access, and directly relevant to embedded devices that could be present in the identified smart grid context. Each previously identified manufacturer is scrutinized to attempt to identify as much relevant firmware as possible. For each of these manufacturers we perform similar procedures. We first utilize popular FTP search engines, such as Napalm FTP Indexer [215], to see if any public corporate FTP servers could be leveraged. We then create a simple website scraper using Scrapy [160] and run it on specific parts of each manufacturers website. Finally, we perform a manual inspection for dynamically generated websites, this applies mostly to each manufacturers download centre. All retrieved images are then filtered for relevance to the current smart grid context.

Table 26: Identified Relevant Smart Grid Firmware

Manufacturer	Number
ABB	18
Aclara	0
Cisco	183
EI	0
Elster	0
GE	5
Honeywell	30
Itron	0
Landis+Gear	0
NI	$2,298$
SE	71
SEL	12
Sensus	0
Siemens	11

Several difficulties have been encountered during this process. Firstly FTP search tools only yield relevant results for Cisco firmware. Most manufactures do not have any documents indexed or only have product documentation. Most manufacturers, such as Itron and Landis+Gear, do not make their firmware publicly available. In most cases it is necessary to possess a scrutinized account or be a customer. On the other hand, some manufactures like ABB or Schneider Electric, have varying degrees of available firmware on their websites or download centers which require no authentication. Nevertheless, there are not vast quantities of firmware publicly available for the majority of the manufactures. Nonetheless, we have been able to extract $2,628$ firmware packages from the mentioned vendors. Table 26 gives an overview of how many images were acquired for each manufacturer.

Within the context of all acquired firmware images, there is a wide range of utilized CPU architectures. The distribution of these is shown in Figure 38. The top ranking results include ARM (82%) and PowerPC (9%). This information, though not conclusive, leads to the credence of the importance of the ARM architecture within the IED context.

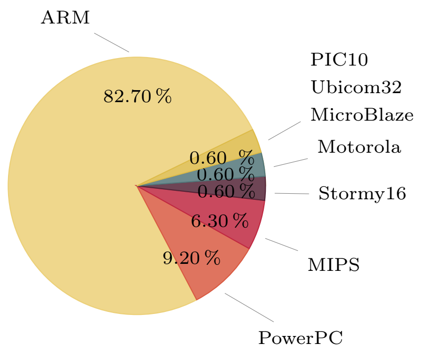


Figure 38: IED Firmware Architecture Distribution

5.1.5 Dataset III - General Dataset

In order to perform all general-purpose evaluations, it is necessary to acquire a large corpus of general-purpose functions. These are utilized for evaluations such as scalability and efficiency. As such, all leftover non-vulnerable functions contained inside compiled libraries utilized in the creation of the first dataset have been reused. This results in a general-purpose corpus that is composed of a total of $3,270,165$ functions.

5.2 Experiments

In order to validate the proposed approach and implementation, it is necessary to perform a set of experimental evaluations and analysis of acquired results. In order to provide a reproducible performance benchmark, the setup of the evaluation environment is first discussed. This is followed by set of use cases that aim at evaluating BinArm's capabilities in real-world scenarios along with its comparison with other state-of-the-art solutions. As such, test cases include function identification accuracy, efficiency and scalability. The last set of tests aim at validating the impact of our multi-staged comparison approach with respect to accuracy and efficiency.

5.2.1 Experimental Setup

All of our experiments are conducted on machines running Windows 7 and Ubuntu 15.04 with Core $i7$ 3.4 GHz CPU and 16GB RAM. BinArm is written in C++ and utilizes a Apache Cassandra to store all source and target functions along with their CFGs and all other extracted features. It exposes a REST interface that can be consumed by another web or desktop application to perform function indexing or comparison.

Vagrant [208] is used to create a specialized environment used for firmware reverse engineering as well as library cross compilation for the ARM architecture. The utilized cross compiler is gcc-arm-linux-gnueabi version 4.7.3 using the debug flag (e.g., -g), and all compatible optimization flags. IDA Pro is one of the main frameworks used when reverse engineering or disassembling firmware and libraries. A custom Python script is used in tandem with IDA Pro to extract functions and their CFGs in the desired JSON format. Docker is used to create a containerized version of the CVE database [214] and its associated search tools [216]. This facilitates its deployment, reusability and usage from within any environment. Python is used in conjunction with the containerized CVE database to extract relevant CVE information related to identified libraries.

5.2.2 Evaluation Metrics

In order to properly evaluate the accuracy of our approach, a specific metric is required. Thus when a search is performed, the utility of the returned results is measured through the total accuracy metric as defined in Equation 9. It is computed by leveraging several different rates namely (i) True Positives (TP); how many relevant functions are retrieved, (ii) False Positives (FP); how many irrelevant functions are retrieved, (iii) True Negatives (TN); how many irrelevant functions are not retrieved, and (iv) False Negatives (FN); how many relevant functions are not retrieved.

$$TA = \frac{TP + TN}{TP + TN + FP + FN} \tag{9}$$

5.2.3 Accuracy

The core functionality proposed by BinArm revolves around its capability to identify similar functions across different binaries. In order to assess this functionality, the proposed total accuracy metric is leveraged. A chosen set of binaries from our general-purpose dataset is selected. This is done by repeatedly randomly selecting 10% of libraries as targets and leveraging the remainder as references. The results of all accuracy tests are presented in Table 27. These conclude that the average total accuracy is equal to 0.92. The lowest scores come from the ntp, and zlib libraries. Through these results it is possible to observe that detection accuracy will vary based on the inter-version variation of diverse projects. Due to random libraries selection process, version variation can cause drops in accuracy.

Table 27: Total Accuracy Results

Library	Total Accuracy
glibc	0.96
libcurl	0.93
libxml2	0.89
lighttpd	0.92
ntp	0.87
openssh	0.89
openssl	0.93
postgresql	0.98
zlib	0.89
Average	**0.92**

5.2.4 Efficiency

There are two main aspects of the BinArm framework that are related to efficiency, namely binary indexing and comparison. Each of these needs to be measured in order to validate the performance of the solution. As such, when indexing a given binary (e.g., vulnerabilities, libraries and firmware images), we measure the time taken. The resulting value includes all overhead generated by feature extraction, basic block weight calculation and indexing. However, the time taken to disassemble binaries in IDA Pro is discarded because (i) being a separate framework its efficiency is independent of BinArm's, and (ii) disassembly time can be approximately evenly distributed over all functions in a binary file. Binary comparison efficiency measurements include the times for feature extraction, and all phases of the multi-staged detection algorithm analysis.

5.2.4.1 Indexing

A first simple baseline evaluation is performed by recording the indexing time for all $3,270,165$ functions inside our general dataset. The sorted results of this process can be seen in Figure 39. A few interesting statistics are: (i) the average indexing time is 0.030 ms, (ii) the smallest indexing time is 0.002 ms for the inflateReset function in zlib, and (iii) the largest indexing time is $119,875.325$ ms to index the base_yyparse function in postgresql. It is also interesting to note that 99.75% of functions are indexed in less than one second.

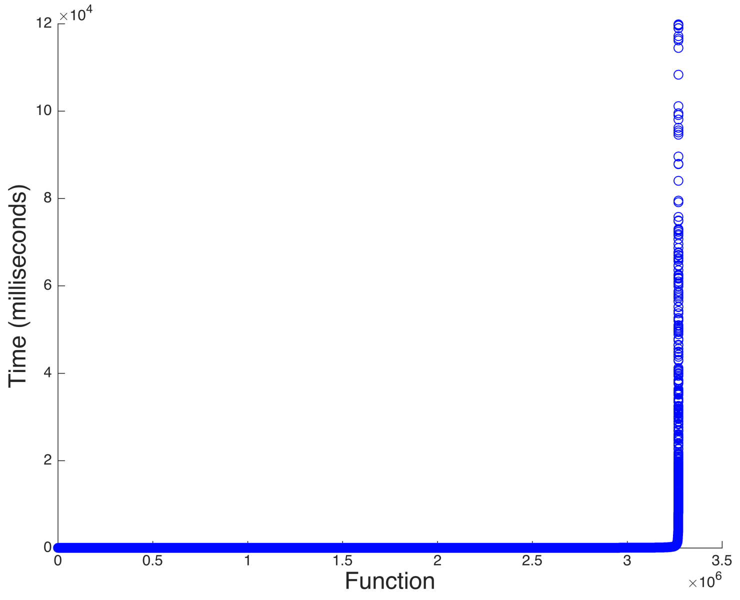


Figure 39: General Dataset Indexing Times

5.2.4.2 Searching

In order to measure the general efficiency capabilities of BinArm, we first leverage our 235 unique vulnerable functions present inside out vulnerability database. These are compared against two different and large corpora of functions in order to measure the time requirements to perform such a task. The first, in order to perform a large-scale evaluation that tests the limits of our framework, leverage all $3,270,165$ functions inside our general dataset. The second, in order to remain semi comparable to other work and provide a general idea of how BinArm performs in the task of firmware analysis, Netgear ReadyNAS 6.1.6 is reused. All results are presented in Figure 40, where the x-axis represents the percentage of number of functions, and the y-axis shows the Cumulative Distribution Function (CDFs) of search time. The observed average function search time inside the general dataset is 0.01 seconds and for the firmware image is 0.008 seconds.

An observation can be made here based on the performance of BinArm. Its efficiency is firmly related to the complexity of the provided target function. This is due to the performance of the final fuzzy matching stage. When given a target function, it is only compared to those identified as similar. As such, when a very large target function is provided, it is only compared to other similarly large ones, which increases search time.

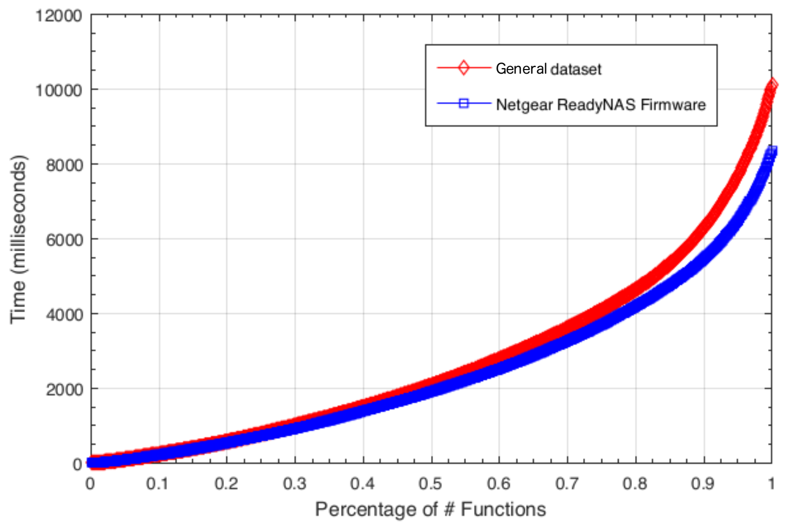


Figure 40: Vulnerable Function Search Time CDF

5.2.5 Scalability

It is necessary to ensure that BinArm is able to handle a large enough volume of functions to manage IED firmware indexing and searching in a reasonable time frame. Furthermore, it must demonstrate that it does not suffer from large performance degradation due to the increased load. Therefore, measurements are considered cumulatively, as opposed to individually, to give an idea of general real-time performance. These results can also provide insight towards assessing usability of the framework.

5.2.5.1 Indexing

The purpose of this experiment is to get an overview of the time required to index a large volume of firmware functions. This is achieved by randomly selecting 1 million functions from our firmware dataset, indexing them, and recording the time it took to do so. The results of this process are presented and further discussed in [150]. It illustrates the CDF of the preparation time for the randomly selected functions. A few interesting statistics are: (i) the median indexing time is 0.008 seconds, (ii) the average function indexing time is 0.02 seconds, and (iii) the vast majority of functions are indexed in under 0.1 seconds.

5.2.5.2 Searching

The experiment aims at evaluating the overall time it takes to perform a large-scale function search operation. To perform this benchmark, we first randomly select a set of $10,000$ target functions from our general dataset. Four other reference sets of incrementing sizes are then randomly selected, each of which is used to perform separate tests. The sizes of these range from $500,000$ to 2 million in steps of half a million. The results of these 4 tests are presented and further discussed in [150].

5.2.6 Impact of Detection Stages

It is desired to measure the impact our proposed detection stages have on the overall detection process. Fundamentally, it is first the question of ensuring that each stage actually decreases overall run-time while keeping a constant accuracy. In order to validate the performance of each stage, four tests are performed leveraging the vulnerability dataset. Each one handles a different activation combination of the shape-based, and the branch-based detections phases. For each run, both the accuracy and efficiency are measured, and then compared. All results are presented and discussed in [150]. These demonstrate that irrespective of which stage is enabled, the overall accuracy remains the same. It is the case however that the overall efficiency is greatly increased by the use of both stages. This is especially the case for the shape-based detector, as it is far less computationally expensive to perform.

5.2.7 Impact of Parameters

It is desired to measure the impact that the values of λ and γ have on both detectors. This is necessary in order to validate that the total accuracy is not reduced by bad parameter value selection.

5.2.7.1 Shape-Based Threshold

It is essential to experimentally validate the identified threshold of λ as discussed in Section 4.4.1.4. First, we disable the branch-based detector in order to prevent any interference with the results. Second, we randomly select 10% of the libraries in our general dataset and leverage them as targets. These are successively compared to the remaining functions with a varying value of λ. It starts off at a value of 5 and is incremented in steps of 5. The results of this process are presented in Figure 41. It demonstrates that a value of 25 is the most appropriate. This assists in confirming our initial selection of λ.

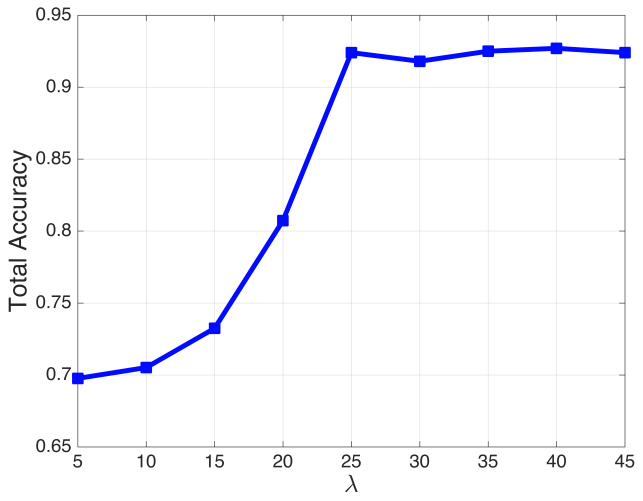


Figure 41: Impact of λ

5.2.7.2 Branch-Based Threshold

We perform a similar evaluation process in order to empirically validate the identified value of γ. First, we disable the shape-based detector so that only the branch-based one is applied. Second, we perform the same target and reference selection process as described above. In incrementing value of λ is initially set to 30% and increases in steps of 5%. The results of this process are presented in Figure 42. They demonstrate that a value of 50% is the most appropriate.

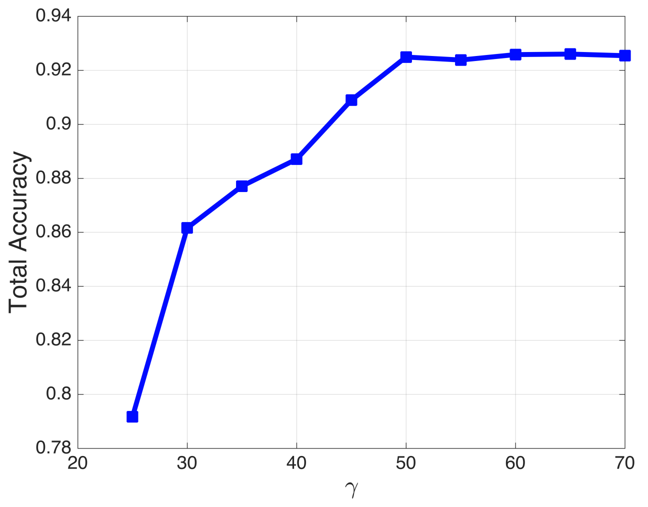
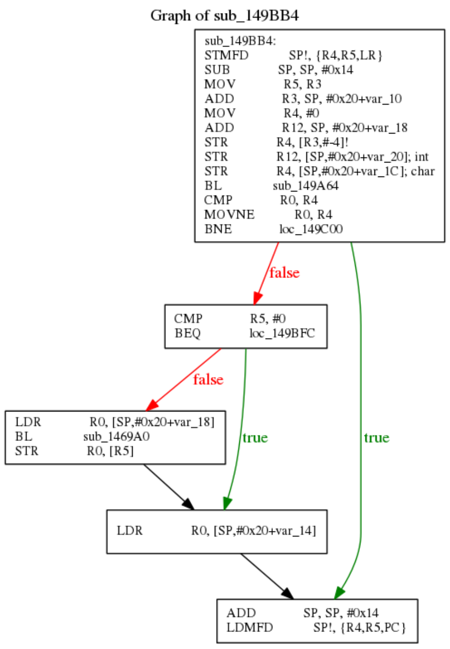


Figure 42: Impact of γ

5.2.8 Firmware Vulnerability Detection

The core functionality of BinArm revolves around its vulnerability detection capabilities. As such, the purpose of this scenario is to leverage it in order to identify vulnerabilities inside of real-world smart grid firmware. First, in order to promote comparison with other methodologies (Subsection 5.2.9.2) we select the ReadyNAS v6.1.6 firmware image. Second, we randomly select four others from our firmware database namely (i) NI PMU 1.0.11, (ii) Schneider Link 150, (iii) Schneider M251, and (iv) Honeywell RTU 150. These are fully disassembled, analyzed, and indexed into our framework. The contents of each of these is then compared to the entire vulnerability database in order to identify potential matches. This generates a set of ranked function matching pairs. A result is considered a potential valid match if its score is greater or equal to 80%. Some of the most interesting results of this analysis are presented in [150].

These results clearly demonstrate BinArm's capability of identifying vulnerable functions within smart-grid firmware. For instance, the the NI PMU 1.0.11 firmware image seemingly contains a critical heap-overflow exploit in the form of CVE-2016-7167. However, its match is not identical at 0.91 and has to be further examined. This is done by examining the differences in their respective CFGs at an instructional and structural level. These two graphs can be seen in Figure 43. After inspection it is clear that differences are quite small and that the vulnerability is most likely present.



```
sub_149BB4:
STMFD     SP!, {R4,R5,LR}
SUB       SP, SP, #0x14
MOV       R5, R3
ADD       R3, SP, #0x20+var_10
MOV       R4, #0
ADD       R12, SP, #0x20+var_18
STR       R4, [R3,#-4]!
STR       R12, [SP,#0x20+var_20]; int
STR       R4, [SP,#0x20+var_1C]; char
BL        sub_149A64
CMP       R0, R4
MOVNE     R0, R4
BNE       loc_149C00
```

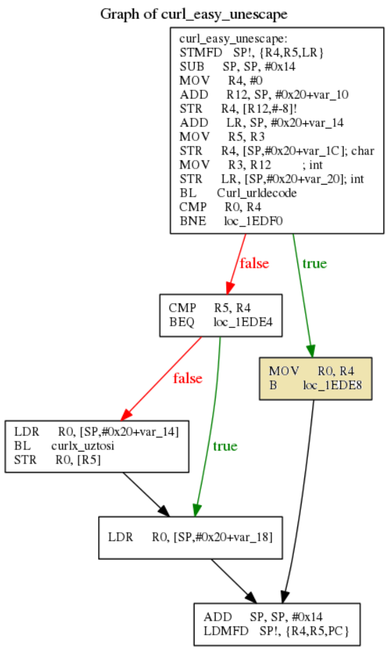
```
CMP       R5, #0
BEQ       loc_149BFC
```

```
LDR       R0, [SP,#0x20+var_18]
BL        sub_1469A0
STR       R0, [R5]
```

```
LDR       R0, [SP,#0x20+var_14]
```

```
ADD       SP, SP, #0x14
LDMFD     SP!, {R4,R5,PC}
```

(a) CFG of `sub_149BB4` in `NI PMU` 1.0.11 firmware



```
curl_easy_unescape:
STMFD   SP!, {R4,R5,LR}
SUB     SP, SP, #0x14
MOV     R4, #0
ADD     R12, SP, #0x20+var_10
STR     R4, [R12,#-8]!
ADD     LR, SP, #0x20+var_14
MOV     R5, R3
STR     R4, [SP,#0x20+var_1C]; char
MOV     R3, R12         ; int
STR     LR, [SP,#0x20+var_20]; int
BL      Curl_urldecode
CMP     R0, R4
BNE     loc_1EDF0
```

```
CMP     R5, R4
BEQ     loc_1EDE4
```

```
MOV     R0, R4
B       loc_1EDE8
```

```
LDR     R0, [SP,#0x20+var_14]
BL      curlx_uztosi
STR     R0, [R5]
```

```
LDR     R0, [SP,#0x20+var_18]
```

```
ADD     SP, SP, #0x14
LDMFD   SP!, {R4,R5,PC}
```

(b) CFG of `curl_easy_unescape` in `libcurl` $v7.50.2$ with CVE-2016-7167

Figure 43: An Example of Function Reuse in IED Firmware

5.2.9 Comparison

5.2.9.1 Indexing

In an effort to promote comparability, several recent works (e.g., Genius, DiscovRe and Muti-MH) have reused the same firmware image as a performance benchmark. The evaluation of interest is the indexing time of an ARM based firmware image, namely ReadyNAS 6.1.6 [217]. This image contains a total of 1,510 binaries that when disassembled result in a total of 2,927,857 basic blocks. As an indexing efficiency comparison, they record the required time each of the different approaches takes to process the chosen firmware image. In essence, we perform the same experiment and present all results, along with those from the other works, in Figure 44. The only methodology that slightly surpasses BinArm in this task is DiscovRe. The similarity is due to the fact that both extraction processes are relatively similar in that they consider CFG extraction time. However, their core difference is that BinArm performs extra computations in the form of feature extraction and path weight calculation.

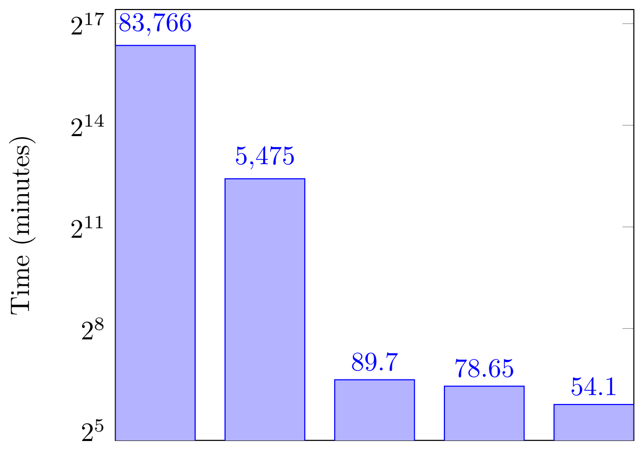


Figure 44: State-of-the-Art Indexing Time

5.2.9.2 Firmware Vulnerability Detection

Similarly to the indexing comparison, other work (e.g., Genius, DiscovRe, Muti-MH and Centroid) specifically attempt to identify CVE-2014-0160 inside the ReadyNAS v6.1.6 firmware image. It represents the well-known Heartbleed vulnerability and is used as a framework capability demonstration. As such, we perform the same identification process with the vulnerable functions (e.g., tls1_process_heartbeat and dtls1_process_heartbeat) and record the search time as presented in Table 28. This is another result that demonstrates BinArms applicability in a real-world firmware analysis scenario.

Table 28: Firmware Vulnerability Identification Comparison

Framework	Time (ms)
BinArm	0.064
Centroid [218]	0.0014
DiscovRe [20]	0.4
Genius [120]	0.0018
Multi-k-MH [21]	1000
Multi-MHTLS [21]	300

5.2.9.3 Qualitative Comparison with Gemini

Gemini [19] is one of the latest and most interesting iterations in binary function similar identification. Its clearly novel approach, that leverages graph embedding in conjunction with siamese neural networks, puts into question the utility of BinArm. At a high level, both BinArm and Gemini's purposes can appear to be quite similar. However, there is a clear contrast in which Gemini performs as a general-purpose methodology, and BinArm, which is a specialized framework for vulnerability detection. Since these purposes overlap, Gemini and BinArm can be compared on their vulnerability identification capabilities. However, a direct comparison between the two is not possible since their implementation is not publicly available. Despite this fact, an analysis that focusses on the comparison of their proposed methodologies, datasets, and results from a realistic firmware vulnerability identification point of view is still possible.

In order to compare two given functions, the authors extract Attributed CFG (ACFG) including seven set of features, and then feed them into a siamese neural network. One of the main objectives is to address the efficiency issues of Genius. The reported experimental results illustrate efficiency improvements over Genius. However, the required training time, which is performed on a powerful server with impressive capabilities (Intel Xeon $E5-2620v4$ 32 core CPU, 96 GB memory, GeForce GTX 1080 GPU), is significant compared to BinArm. Additionally, the proposed multi-stage detection algorithm in BinArm prevents the pairwise comparison and speeds up the process.

Gemini solely relies on the use of a siamese neural network to perform their comparison. Although novel, it has several limitations. Their comparison process ultimately relies on a few basic features, which can clearly be improved. This can be seen when comparing them to BinArm's rigorous feature identification, evaluation, and selection. Such feature choices are reflected though their reported vulnerability identification accuracy of about 82%. This is far less when in comparison to BinArm's 92%. This can be partially explained by BinArm's multi-phased detection process which takes into account a far larger scope of information relative to a given function. For example, they barely take into account function semantics by only counting the number of arithmetic instructions. It is often the case that a vulnerability is due to a few faulty instructions, which cannot be detected without taking function semantics into account. This is highly in contrast to BinArm's branch-based detection phase, where each basic blocks instruction probability densities are leveraged as well as the features employed in fuzzy matching stage.

When updating the embedding for a given basic block in a graph, they take neighboring blocks into account. This is done by summing their values together. This process however does not take into account the parent and child relationships between the basic blocks. This will slightly impact their overall accuracy. On the other hand, in BinArm's branch-based detection phase, neighboring blocks relationships are considered. Parents are joined to the current weight vector and children are combined.

In the context of vulnerability detection, it is necessary to be able to quickly index new firmware images and vulnerabilities to perform assessments. Thus, the time required to constantly retrain the siamese neural network and re-generate the embeddings is a major disadvantage. Although in a lab setting, taking an entire day to perform this entire process for a single new vulnerability may be acceptable, this is not necessarily the case in a real-world scenario. When a new vulnerability or firmware is released, experts should not have to wait for long periods of time before being able to perform their work. Therefore, BinArm greatly outperforms Gemini with respect to the indexing of vulnerable functions or firmware into the system.

One last difference of note is the size of vulnerability datasets presented in the two works. Gemini has a total of 154 vulnerable functions and presents a simple use case that employs two of them. BinArm's vulnerability dataset contains $5,103$ vulnerable functions, all of which are used for vulnerability identification. Moreover, for each CVE the Gemini needs to be trained separately and few times, while BinArm indexes the relevant vulnerable functions once and together.

Chapter 6

Conclusion

The smart grid critical infrastructure is highly dependent on the well being of all of its components. Modern deployments heavily rely on the advanced capabilities that IEDs provide. These devices' firmware images often contain known vulnerabilities that could be exploited by attackers resulting in devastating impacts. This in turn would cause devastating effects on national economic security and national safety. As such a scalable domain specific approach is required in order to assess the security of IED firmware. This is in contrast to other existing general-purpose techniques that lack specialization. This is demonstrated through the absence of a set of IED specific vulnerabilities and their inability to scale to the size of IED firmware.

A foremost step in protecting these critical infrastructures is the analysis of their firmware from a security perspective. In this work we proposed our take on this solution in the form of BinArm. It is a scalable and efficient vulnerable function detection framework specifically designed for the evaluation of IED firmware. In order to perform its analysis, it leveraged substantial datasets of relevant smart grid vulnerabilities and firmware images. Its analysis procedure is composed of a multi-stage detection methodology that leverages function syntactic, semantic, structural and statistical features in order to identify vulnerabilities. It is implemented by taking a coarse-to-fine grained multi-stage function matching approach. As such it (i) first filters out dissimilar functions based on a group of heterogeneous features, (ii) it then further filters out dissimilar functions based on their execution paths, and then (iii) finally identifies candidate functions based on fuzzy graph matching.

The shape based phase relied upon a set of extracted heterogeneous features that represent a given function. These were used to prune the initial search space. First, a set of relevant features had be identified. Second, they had to be classified in order to extract only the most relevant ones. Third, the extracted features were used to calculate a relevance threshold. It was used to determine if a given function was similar enough to be further analyzed.

The second phase compares functions from a different informational standpoint. As such, it assumes that similar functions will have similar execution paths. Thus, the branch-based phase compares the paths of the two provided function CFGs. This is achieved by leveraging a weighted path comparison algorithm to assess the distance between the two graphs. Just as in the shape-based detection phase, if the resulting distance is over a given threshold, the reference function is removed from the pool of possible candidate functions.

All functions that remain after the fist two stages are lastly compared using a fuzzy matching-based approach. These have already been assessed to have relatively high similarity with the target and merit to be fully analyzed. Each function is compared using a combination of the LCS and Hungarian algorithms. The results of this phase are ordered pairs of the highest identified matches.

In order to perform all benchmarks and evaluations, substantial datasets needed to be compiled. The first is to provide a set of target firmware. This set is analyzed to determine the presence or absence of known security vulnerabilities. The second is to provide a set of reference vulnerabilities. These are specific functions that have been acquired due to their relation with the desired smart grid scope of the work and the presence of one or more known vulnerabilities. The third is composed of general-purpose binary functions. All datasets leverage acquired knowledge relevant to prominent smart grid manufacturers and the open-source software they leverage.

In order to validate BinArm's capabilities, it was placed through a rigorous set of evaluations. As such, test cases included function identification accuracy, efficiency, scalability. This is followed by set of use cases that aim at evaluating BinArm's capabilities in real-world scenarios along with its comparison with other State Of The Art solutions. These demonstrated that BinArm can identify vulnerabilities within a given IED firmware with a total accuracy of 0.92. These further established that the proposed methodology outperforms other fuzzy matching techniques by three orders of magnitude. The evaluation resulted in the successful identification of 93 potentially vulnerable functions inside of real-world smart grid IED firmware.

6.1 Contributions

We have identified the main contributions that this work provides as follows:

- To the best of our knowledge, we have produced the first large-scale vulnerability database that is specifically tailored for IED device firmware. It covers all major manufacturers (e.g., Siemens, ABB, Schneider, SEL, etc.). At the same time, we have created the first IED firmware dataset that is specifically related to the smart grid.

- We proposed a multi-stage vulnerability detection framework entitled BinArm. This approach addressed the scalability issues of previous work. It also outperformed existing fuzzy matching approaches with respect to speed, while maintaining accuracy. Our experiments demonstrated that BinArm is three orders of magnitude faster than existing fuzzy matching approaches.

- We proposed the Weighted Normalized Tree Distance (WNTD) metric. It enabled the comparison of execution paths between two given functions. It did so by comparing their respective acyclical CFG paths by taking into account a weight for each basic block in a given path. Weights are generated based on a basic blocks code contents, and relative position within the overall CFG.

- Rigorous experiments and benchmarks were performed on BinArm, which validated its performance both from an efficiency and accuracy standpoint. These evaluations demonstrated that the framework can identify similar functions with a total accuracy of 0.92.

- The capabilities of BinArm were validated by performing meticulous experiments on real work IED firmware images. This process identified 93 potential CVEs inside these images, the majority of which have been confirmed using manual analysis.

6.2 Limitations

All performed experiments enabled the identification of several limitations within BinArm. These are further outlined and discussed.

Function Inlining: This capability is not currently supported. It is partially due to the fact that if a small target function is compared to a larger reference function that contains it, the fuzzy matching algorithm will likely identify the similarities. However, due to our multi-stage approach that favors the final analysis of functions that have a similar shape, this will most likely never happen. Furthermore, the true limitation lies in matching a large target function that contains a smaller reference function. Due to the use of the longest path in the final analysis stage, it is impossible to identify the match.

Multiple Architectures: A large amount of IEDs deployed in the smart grid context utilize an ARM based processor. Due to this fact, it is a design choice to create a framework that specifically focusses on the ARM architecture. However, there are still devices that leverage other platforms (e.g., PowerPC, MIPS) that are obviously excluded.

6.3 Future Work

In order to further the capabilities and mitigate the limitations of BinArm, three primary directions for future work have been identified. The first interesting improvement is the introduction of multi-platform support. This will widen the scope of smart-grid firmware images that can be analyzed for vulnerable functions. The second is the mitigation of the partial function inlining limitation. This problem is highly common in the state of the art and would be a strong contribution. Finally, it would be interesting to propose a dynamic analysis framework that would leverage BinArm's results as inputs. Its purpose would be to perform dynamic analysis on the resulting function pairs in order to assess whether they do have similar execution behaviors. It could also be used to detect run-time data-oriented exploits.

Bibliography

[1] ICS-CERT: Critical Infrastructure Sectors (2017). https://www.dhs.gov/critical-infrastructure-sectors. Accessed: 2018.

[2] Nicolas Falliere, Liam O Murchu, and Eric Chien. W32. stuxnet dossier. *White paper, Symantec Corp., Security Response*, 5(6), 2011.

[3] Ralph Langner. Stuxnet: Dissecting a cyberwarfare weapon. *IEEE Security & Privacy*, 9(3):49–51, 2011.

[4] Jose Nazario. Blackenergy DDoS bot analysis. *Arbor Networks*, 2007.

[5] Ukraine's Power Outage Was a Cyber Attack: Ukrengo. https://www.reuters.com/article/us-ukraine-cyber-attack-energy/ukraines-power-outage-was-a-cyber-attack-ukrenergo-idUSKBN1521BA. Accessed: 2018.

[6] WIN32/INDUSTROYER A new threat for industrial control systems. https://www.welivesecurity.com/wp-content/uploads/2017/06/Win32_Industroyer.pdf. Accessed: 2018.

[7] Crashoverride: Analysis of the Threat to Electric Grid Operations. https://dragos.com/blog/crashoverride/CrashOverride-01.pdf. Accessed: 2018.

[8] Industroyer: Biggest threat to industrial control systems since Stuxnet. https://www.welivesecurity.com/2017/06/12/industroyer-biggest-threat-industrial-control-systems-since-stuxnet/. Accessed: 2018.

[9] Security Intelligence (2017). https://securityintelligence.com/attackstargeting-industrial-control-systems-ics-up-110-percent/. Accessed: 2018.

[10] Innovation Series. Business Blackout. 2015.

[11] Ruijin Zhu, Baofeng Zhang, Junjie Mao, Quanxin Zhang, and Yu-an Tan. A methodology for determining the image base of arm-based industrial control system firmware. *International Journal of Critical Infrastructure Protection*, 16:26–35, 2017.

[12] Arm Holdings plc - Roadshow Slides. `https://www.arm.com/-/media/global/company/investors/PDFs/Arm_SB_Q3_2017_Roadshow_Slides_Final.pdf`. Accessed: 2018.

[13] YooJin Kwon, Huy Kang Kim, Koudjo M Koumadi, Yong Hun Lim, and Jong In Lim. Automated vulnerability analysis technique for smart grid infrastructure. In *Power & Energy Society Innovative Smart Grid Technologies Conference (ISGT), 2017 IEEE*, pages 1–5. IEEE, 2017.

[14] Binbin Chen, Xinshu Dong, Guangdong Bai, Sumeet Jauhar, and Yueqiang Cheng. Secure and efficient software-based attestation for industrial control devices with arm processors. In *ACSAC*, 2017.

[15] Christopher Kruegel, William Robertson, Fredrik Valeur, and Giovanni Vigna. Static disassembly of obfuscated binaries. In *USENIX security Symposium*, volume 13, pages 18–18, 2004.

[16] Muqing Liu, Yuanyuan Zhang, Juanru Li, Junliang Shu, and Dawu Gu. Security analysis of vendor customized code in firmware of embedded device. In *International Conference on Security and Privacy in Communication Systems*, pages 722–739. Springer, 2016.

[17] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. Firmalice-Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware. In *NDSS*, 2015.

[18] Qian Feng, Minghua Wang, Mu Zhang, Rundong Zhou, Andrew Henderson, and Heng Yin. Extracting conditional formulas for cross-platform bug search. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pages 346–359. ACM, 2017.

[19] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. Neural Network-based Graph Embedding for Cross-Platform Binary Code Similarity Detection. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 363–376. ACM, 2017.

[20] Sebastian Eschweiler, Khaled Yakdan, and Elmar Gerhards-Padilla. discovRE: Efficient Cross-Architecture Identification of Bugs in Binary Code. In *NDSS*, 2016.

[21] Jannik Pewny, Behrad Garmany, Robert Gawlik, Christian Rossow, and Thorsten Holz. Cross-architecture bug search in binary executables. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 709–724. IEEE, 2015.

[22] Paria Shirani, Lingyu Wang, and Mourad Debbabi. BinShape: Scalable and Robust Binary Library Function Identification Using Function Shape. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 301–324. Springer, 2017.

[23] Yichen Zheng, William Ott, Chinmaya Gupta, and Dan Graur. A scale-free method for testing the proportionality of branch lengths between two phylogenetic trees. 2015.

[24] Thomas H Cormen. *Introduction to algorithms.* MIT press, 2009.

[25] James Munkres. Algorithms for the assignment and transportation problems. 1957.

[26] Common Vulnerabilities and Exposures. `https://nvd.nist.gov/`. Accessed: 2018.

[27] Blackout Diagram. `http://3.bp.blogspot.com/-CHCAa7bwK-8/VXW8BlhSnHI/AAAAAAAADHc/YnzlUSxQ1R8/s1600/blackout_diagram.png`. Accessed: 2018.

[28] Frances Cleveland. IEC TC57 WG15: IEC 62351 security standards for the power system information infrastructure. *White Paper*, 2012.

[29] Vehbi C Gungor, Dilan Sahin, Taskin Kocak, Salih Ergut, Concettina Buccella, Carlo Cecati, and Gerhard P Hancke. Smart grid technologies: Communication technologies and standards. 2011.

[30] CLP: Power Transmission and Distribution in the Smart Grid. `https://www.clp.com.hk/en/about-clp/power-transmission-and-distribution/smart-grid`. Accessed: 2018.

[31] Steven W Blume. *Electric power system basics for the nonelectrical professional.* John Wiley & Sons, 2016.

[32] IEC 61850 - Communication networks and systems for power utility automation. `https://webstore.iec.ch/publication/6028`. Accessed: 2018.

[33] RE Mackiewicz. Overview of IEC 61850 and Benefits. In *Power Systems Conference and Exposition, 2006. PSCE'06. 2006 IEEE PES*, pages 623–630. IEEE, 2006.

[34] Steffen Fries, Hans Joachim Hof, and Maik Seewald. Enhancing IEC 62351 to improve security for energy automation in smart grid environments. In *Internet and Web Applications and Services (ICIW), 2010 Fifth International Conference on*, pages 135–142. IEEE, 2010.

[35] Overview of IEC 61850 networks. https://www.kth.se/social/files/57fb7facf276540784e68ea7/Substation%20Communication%20Protocols.pdf. Accessed: 2018.

[36] IEC 61850 Substation Overview. https://www.moxa.com/doc/guidebooks/IEC_61850_Substation_Overview.pdf. Accessed: 2018.

[37] IEC61850 Logical Nodes. http://www.nettedautomation.com/standardization/IEC_TC57/WG10-12/iec61850/models/LNs_2012-07-20_o.pdf. Accessed: 2018.

[38] How to use IEC 61850 in protection and automation. https://library.e.abb.com/public/fed26a71538479c3c12570d50034fbe4/Rapport.pdf. Accessed: 2018.

[39] Annabelle Lee and Tanya Brewer. Smart grid cyber security strategy and requirements. *Draft Interagency Report NISTIR*, 2009.

[40] S John. CIA Says Hackers Attack Global Power Grid. *Info. Security*, 2009.

[41] Yong Wang, Da Ruan, Dawu Gu, Jason Gao, Daming Liu, Jianping Xu, Fang Chen, Fei Dai, and Jinshi Yang. Analysis of smart grid security standards. In *Computer Science and Automation Engineering (CSAE), 2011 IEEE International Conference on*, volume 4, pages 697–701. IEEE, 2011.

[42] EPRI. http://smartgrid.epri.com/Index.aspx. Accessed: 2018.

[43] Karl Christoph Ruland, Jochen Sassmannshausen, Karl Waedt, and Natasa Zivic. Smart grid security–an overview of standards and guidelines. *e & i Elektrotechnik und Informationstechnik*, 2017.

[44] NERC CIP Standards. http://www.nerc.com/pa/Stand/Pages/CIPStandards.aspx. Accessed: 2018.

[45] NERC CIP. http://searchcompliance.techtarget.com/definition/NERC-CIP-critical-infrastructure-protection. Accessed: 2018.

[46] Karl Christoph Ruland and Jochen Sassmannshausen. Non-repudiation Services for the MMS Protocol of IEC 61850. In *International Conference on Research in Security Standardisation*, pages 70–85. Springer, 2015.

[47] IEC 62351. https://webstore.iec.ch/publication/6912. Accessed: 2018.

[48] NESCOR. http://smartgrid.epri.com/NESCOR.aspx. Accessed: 2018.

[49] A Lee. Electric sector failure scenarios and impact analyses-Draft. *National Electric Sector Cybersecurity Organization Resource (NESCOR) Technical Working Group*, 1, 2013.

[50] IEC 62351:2018 SER Series. `https://webstore.iec.ch/publication/6912`. Accessed: 2018.

[51] Zhuo Lu, Xiang Lu, Wenye Wang, and Cliff Wang. Review and evaluation of security threats on the communication networks in the smart grid. In *Military Communications Conference, 2010-MILCOM 2010*, pages 1830–1835. IEEE, 2010.

[52] Nishchal Kush, Ejaz Ahmed, Mark Branagan, and Ernest Foo. Poisoned GOOSE: exploiting the GOOSE protocol. In *Proceedings of the Twelfth Australasian Information Security Conference-Volume 149*, pages 17–22. Australian Computer Society, Inc., 2014.

[53] Juan Hoyos, Mark Dehus, and Timthy X Brown. Exploiting the GOOSE protocol: A practical attack on cyber-infrastructure. In *Globecom Workshops (GC Wkshps), 2012 IEEE*, pages 1508–1513. IEEE, 2012.

[54] Alfonso Valdes, Cui Hang, Prosper Panumpabi, Nitin Vaidya, Chris Drew, and Dimitry Ischenko. Design and simulation of fast substation protection in IEC 61850 environments. In *Modeling and Simulation of Cyber-Physical Energy Systems (MSCPES), 2015 Workshop on*, pages 1–6. IEEE, 2015.

[55] Junho Hong, Chen-Ching Liu, and Manimaran Govindarasu. Integrated anomaly detection for cyber security of the substations. *IEEE Transactions on Smart Grid*, 5(4):1643–1653, 2014.

[56] Charalambos Konstantinou, Michail Maniatakos, Fareena Saqib, Shiyan Hu, Jim Plusquellic, and Yier Jin. Cyber-physical systems: A security perspective. In *Test Symposium (ETS), 2015 20th IEEE European*, pages 1–8. IEEE, 2015.

[57] Paria Jokar, Nasim Arianpoo, and Victor Leung. A survey on security issues in smart grids. *Security and Communication Networks*, 9(3):262–273, 2016.

[58] Igor Nai Fovino, Andrea Carcano, Marcelo Masera, and Alberto Trombetta. An experimental investigation of malware attacks on SCADA systems. *International Journal of Critical Infrastructure Protection*, 2(4):139–145, 2009.

[59] Mike Rogers and CA Dutch Ruppersberger. *Investigative report on the US national security issues posed by Chinese telecommunications companies Huawei and ZTE: a report.* US House of Representatives, 2012.

[60] Aldar C-F Chan and Jianying Zhou. Cyber–physical device authentication for the smart grid electric vehicle ecosystem. *IEEE Journal on Selected Areas in Communications*, 32(7):1509–1517, 2014.

[61] Lucie Langer, Florian Skopik, Paul Smith, and Markus Kammerstetter. From old to new: Assessing cybersecurity risks for an evolving smart grid. *computers & security*, 62:165–176, 2016.

[62] Michael LeMay and Carl A Gunter. Cumulative attestation kernels for embedded systems. In *European Symposium on Research in Computer Security*, pages 655–670. Springer, 2009.

[63] Wenye Wang and Zhuo Lu. Cyber security in the smart grid: Survey and challenges. *Computer Networks*, 57(5):1344–1371, 2013.

[64] Introduction to NISTIR 7628 Guidelines for Smart Grid Cyber Security. `https://www.nist.gov/sites/default/files/documents/smartgrid/nistir-7628_total.pdf`. Accessed: 2018.

[65] Kim-Kwang Raymond Choo, Yunsi Fei, Yang Xiang, and Yu Yu. Embedded device forensics and security. *ACM Transactions on Embedded Computing Systems (TECS)*, 16(2):50, 2017.

[66] Maneli Malek Pour, Arash Anzalchi, and Arif Sarwat. A review on cyber security issues and mitigation methods in smart grid systems. In *SoutheastCon, 2017*, pages 1–4. IEEE, 2017.

[67] IEEE 1686: IEEE Standard for Intelligent Electronic Devices Cyber Security Capabilities. `https://ieeexplore.ieee.org/document/6704702/`. Accessed: 2018.

[68] IEEE 1402: Guide for Electric Power Substation Physical and Electronic Security. `https://ieeexplore.ieee.org/document/836296/`. Accessed: 2018.

[69] Scott Davidson and Bruce D Shriver. An overview of firmware engineering. *Computer*, 11(5):21–33, 1978.

[70] IEEE Standards Coordinating Committee et al. IEEE Standard Glossary of Software Engineering Terminology (IEEE Std 610.12-1990). Los Alamitos. *CA: IEEE Computer Society*, 169, 1990.

[71] Boyd L Summers. *Software engineering reviews and audits.* CRC Press, 2011.

[72] Emily Blem, Jaikrishnan Menon, and Karthikeyan Sankaralingam. Power struggles: Revisiting the RISC vs. CISC debate on contemporary ARM and x86 architectures. In *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*, pages 1–12. IEEE, 2013.

[73] Jonas Zaddach and Andrei Costin. Embedded devices security and firmware reverse engineering. *Black-Hat USA*, 2013.

[74] Peter Barry and Patrick Crowley. *Modern embedded computing: designing connected, pervasive, media-rich systems.* Elsevier, 2012.

[75] Shahed E Quadir, Junlin Chen, Domenic Forte, Navid Asadizanjani, Sina Shahbazmohamadi, Lei Wang, John Chandy, and Mark Tehranipoor. A survey on chip to system reverse engineering. *ACM journal on emerging technologies in computing systems (JETC)*, 13(1):6, 2016.

[76] Randy Torrance and Dick James. The state-of-the-art in IC reverse engineering. In *Cryptographic Hardware and Embedded Systems-CHES 2009*, pages 363–381. Springer, 2009.

[77] 7-Zip. `http://www.7-zip.org/download.html`. Accessed: 2018.

[78] XZ. `https://linux.die.net/man/1/xz`. Accessed: 2018.

[79] unYAFFS. `https://github.com/ehlers/unyaffs`. Accessed: 2018.

[80] Binwalk reverse engineering tool. `https://github.com/devttys0/binwalk`. Accessed: 2018.

[81] Bless - Gtk Hex Editor. `https://github.com/bwrsandman/Bless`. Accessed: 2018.

[82] dd - convert and copy a file. `http://man7.org/linux/man-pages/man1/dd.1.html`. Accessed: 2018.

[83] Signsrch signature identification tool. `http://aluigi.altervista.org/mytoolz.htm`. Accessed: 2018.

[84] strings - print the strings of printable characters in files. `https://linux.die.net/man/1/strings`. Accessed: 2018.

[85] Determine file type. `https://linux.die.net/man/1/file`. Accessed: 2018.

[86] IDA Pro. `https://www.hex-rays.com/products/ida/`.

[87] IDA Python. `https://github.com/idapython/src`.

[88] Cullen Linn and Saumya Debray. Obfuscation of executable code to improve resistance to static disassembly. In *Proceedings of the 10th ACM conference on Computer and communications security*, pages 290–299. ACM, 2003.

[89] Andrei Costin, Jonas Zaddach, Aurélien Francillon, Davide Balzarotti, and Sophia Antipolis. A Large-Scale Analysis of the Security of Embedded Firmwares. In *USENIX Security Symposium*, pages 95–110, 2014.

[90] Binwalk - Creating Custom Signatures. https://github.com/ReFirmLabs/binwalk/wiki/Creating-Custom-Signatures. Accessed: 2018.

[91] LZMA. http://www.7-zip.org/sdk.html. Accessed: 2018.

[92] Tar. https://linux.die.net/man/1/tar. Accessed: 2018.

[93] Unrar. https://www.rarlab.com/rar_add.htm. Accessed: 2018.

[94] Binary Analysis Tool (BAT). http://www.binaryanalysis.org/. Accessed: 2018.

[95] GNU - Binary Utilities. https://www.gnu.org/software/binutils/. Accessed: 2018.

[96] cpu_rec: Recognize CPU Instructions In Arbitrary Binary Files. https://github.com/airbus-seclab/cpu_rec. Accessed: 2018.

[97] Object Dump. https://linux.die.net/man/1/objdump.

[98] disassembler.io: Online Disassembler. https://onlinedisassembler.com/. Accessed: 2018.

[99] ERESI Framework. https://github.com/thorkill/eresi. Accessed: 2018.

[100] Hiew. http://www.hiew.ru/. Accessed: 2018.

[101] Hopper v4: The macOS and Linux Disassembler. https://www.hopperapp.com/. Accessed: 2018.

[102] Immunity Debugger: write exploits, analyze malware, and reverse engineer binary files. https://www.immunityinc.com/products/debugger/. Accessed: 2018.

[103] Win32 PE Disassembler: Dig into Executables. http://www.heaventools.com/PE_Explorer_disassembler.htm. Accessed: 2018.

[104] Radare: Portable Reversing Framework. https://rada.re/r/index.html. Accessed: 2018.

[105] Relyze: Interactive Software Analysis. https://www.relyze.com/overview.html. Accessed: 2018.

[106] x64dbg: An open-source x64/x32 debugger for windows. https://x64dbg.com/#start. Accessed: 2018.

[107] Cisco IP phones firmware packer/unpacker. https://github.com/kbdfck/cnu-fpu. Accessed: 2018.

[108] Squash File System Tools. `https://github.com/plougher/squashfs-tools`. Accessed: 2018.

[109] UBI File-System. `http://www.linux-mtd.infradead.org/doc/ubifs.html`. Accessed: 2018.

[110] Hexdump. `https://www.freebsd.org/cgi/man.cgi?query=hexdump&sektion=1`. Accessed: 2018.

[111] WinHex: Computer Forensics, Data Recovery Software, Hex Editor, Disk Editor. `https://www.x-ways.net/winhex/`. Accessed: 2018.

[112] wxHexEditor. `http://www.wxhexeditor.org/home.php`. Accessed: 2018.

[113] Hash Identifier. `https://code.google.com/archive/p/hash-identifier/`.

[114] PEiD: Detects packers, cryptors, and compilers bundled in PE executables. `http://www.softpedia.com/get/Programming/Packers-Crypters-Protectors/PEiD-updated.shtml`. Accessed: 2018.

[115] Signature Search. `http://aluigi.altervista.org/mytoolz.htm`.

[116] TrID - File Identifier. `http://mark0.net/soft-trid-e.html`.

[117] Yaniv David and Eran Yahav. Tracelet-based code search in executables. *ACM SIGPLAN Notices*, 49(6):349–360, 2014.

[118] Yaniv David, Nimrod Partush, and Eran Yahav. Statistical similarity of binaries. *ACM SIGPLAN Notices*, 51(6):266–280, 2016.

[119] Yaniv David, Nimrod Partush, and Eran Yahav. Similarity of Binaries through re-Optimization. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 79–94. ACM, 2017.

[120] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. Scalable graph-based bug search for firmware images. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 480–491. ACM, 2016.

[121] Wei Ming Khoo, Alan Mycroft, and Ross Anderson. Rendezvous: A search engine for binary code. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 329–338. IEEE Press, 2013.

[122] He Huang, Amr M Youssef, and Mourad Debbabi. BinSequence: Fast, Accurate and Scalable Binary Code Reuse Detection. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pages 155–166. ACM, 2017.

[123] Manuel Egele, Maverick Woo, Peter Chapman, and David Brumley. Blanket execution: Dynamic similarity testing for program binaries and components. USENIX, 2014.

[124] Daming D Chen, Maverick Woo, David Brumley, and Manuel Egele. Towards Automated Dynamic Analysis for Linux-based Embedded Firmware. In *NDSS*, 2016.

[125] Thanassis Avgerinos, Sang Kil Cha, Alexandre Rebert, Edward J Schwartz, Maverick Woo, and David Brumley. Automatic exploit generation. *Communications of the ACM*, 57(2):74–84, 2014.

[126] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. Unleashing mayhem on binary code. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 380–394. IEEE, 2012.

[127] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *NDSS*, volume 16, pages 1–16, 2016.

[128] Jonas Zaddach, Luca Bruno, Aurelien Francillon, and Davide Balzarotti. AVATAR: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares. In *NDSS*, 2014.

[129] Andrei Costin, Apostolis Zarras, and Aurélien Francillon. Automated dynamic firmware analysis at scale: a case study on embedded web interfaces. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, pages 437–448. ACM, 2016.

[130] Drew Davidson, Benjamin Moench, Thomas Ristenpart, and Somesh Jha. FIE on Firmware: Finding Vulnerabilities in Embedded Systems Using Symbolic Execution. In *USENIX Security Symposium*, pages 463–478, 2013.

[131] Martial Bourquin, Andy King, and Edward Robbins. Binslayer: accurate comparison of binary executables. In *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop*, page 4. ACM, 2013.

[132] Saed Alrabaee, Paria Shirani, Lingyu Wang, and Mourad Debbabi. Sigma: A semantic integrated graph matching approach for identifying reused functions in binary code. *Digital Investigation*, 12:S61–S71, 2015.

[133] Saed Alrabaee, Paria Shirani, Lingyu Wang, and Mourad Debbabi. FOSSIL: A Resilient and Efficient System for Identifying FOSS Functions in Malware Binaries. *ACM Transactions on Privacy and Security (TOPS)*, 21(2):8, 2018.

[134] Manuel Egele, Theodoor Scholte, Engin Kirda, and Christopher Kruegel. A survey on automated dynamic malware-analysis techniques and tools. *ACM computing surveys (CSUR)*, 44(2):6, 2012.

[135] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2E: A platform for in-vivo multi-path analysis of software systems. *ACM SIGPLAN Notices*, 46(3):265–278, 2011.

[136] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *OSDI*, volume 8, pages 209–224, 2008.

[137] Dorothy E Denning and Peter J Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, 1977.

[138] Anusha Damodaran, Fabio Di Troia, Corrado Aaron Visaggio, Thomas H Austin, and Mark Stamp. A comparison of static, dynamic, and hybrid analysis for malware detection. *Journal of Computer Virology and Hacking Techniques*, 13(1):1–12, 2017.

[139] Brian Chess and Jacob West. *Secure programming with static analysis*. Pearson Education, 2007.

[140] Ashkan Rahimian, Paria Shirani, Saed Alrbaee, Lingyu Wang, and Mourad Debbabi. Bincomp: A stratified approach to compiler provenance attribution. *Digital Investigation*, 14:S146–S155, 2015.

[141] Saed Alrabaee, Noman Saleem, Stere Preda, Lingyu Wang, and Mourad Debbabi. Oba2: An onion approach to binary code authorship attribution. *Digital Investigation*, 11:S94–S103, 2014.

[142] Saed Alrabaee, Paria Shirani, Mourad Debbabi, and Lingyu Wang. On the feasibility of malware authorship attribution. In *International Symposium on Foundations and Practice of Security*, pages 256–272. Springer, 2016.

[143] CVE-2015-4485. `http://www.cvedetails.com/cve/CVE-2015-4485/`. Accessed: 2018.

[144] BinDiff. `http://www.zynamics.com/bindiff.html`. Accessed: 2018.

[145] Diaphora: A Program Diffing Plugin for IDA Pro. `https://github.com/joxeankoret/diaphora`. Accessed: 2018.

[146] PatchDiff2: Binary Diffing Plugin for IDA. `https://code.google.com/p/patchdiff2/`. Accessed: 2018.

[147] Halvar Flake. Graph-based binary analysis. *Blackhat Briefings 2002*, 2002.

[148] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H Witten. The WEKA data mining software: an update. *ACM SIGKDD explorations newsletter*, 11(1):10–18, 2009.

[149] Artem Dinaburg and Andrew Ruef. Mcsema: Static translation of x86 instructions to llvm. In *ReCon 2014 Conference, Montreal, Canada*, 2014.

[150] Paria Shirani, Leo Collard, Basile L Agba, Bernard Lebel, Mourad Debbabi, Lingyu Wang, and Aiman Hanna. Binarm: Scalable and efficient detection of vulnerabilities in firmware images of intelligent electronic devices. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 114–138. Springer, 2018.

[151] Mohammad Reza Farhadi, Benjamin CM Fung, Philippe Charland, and Mourad Debbabi. Binclone: Detecting code clones in malware. In *Software Security and Reliability (SERE), 2014 Eighth International Conference on*, pages 78–87. IEEE, 2014.

[152] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. Sok:(state of) the art of war: Offensive techniques in binary analysis. In *Security and Privacy (SP), 2016 IEEE Symposium on*, pages 138–157. IEEE, 2016.

[153] Angr Framework. `https://github.com/angr/angr`. Accessed: 2018.

[154] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, page 75. IEEE Computer Society, 2004.

[155] Yan Shoshitaishvili. Python bindings for Valgrind's VEX IR, 2014.

[156] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: three decades later. *Communications of the ACM*, 56(2):82–90, 2013.

[157] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, volume 41, page 46, 2005.

[158] David Maynor. *Metasploit toolkit for penetration testing, exploit development, and vulnerability research.* Elsevier, 2011.

[159] Nessus Professional. `http://www.nessus.org`. Accessed: 2018.

[160] Scrapy Web Scraper. `https://scrapy.org/`. Accessed: 2018.

[161] ARMv8 Instruction Set Overview. `https://www.element14.com/community/servlet/JiveServlet/previewBody/41836-102-1-229511/ARM.Reference_Manual.pdf`. Accessed: 2018.

[162] IDA Pro Operand Types. `https://www.hex-rays.com/products/ida/support/idadoc/276.shtml`. Accessed: 2018.

[163] Christopher Griffin. Graph Theory: Penn State Math 485 Lecture.

[164] Babak Bashari Rad, Maslin Masrom, and Suahimi Ibrahim. Opcodes histogram for classifying metamorphic portable executables malware. In *e-Learning and e-Technologies in Education (ICEEE), 2012 International Conference on*, pages 209–213. IEEE, 2012.

[165] Annie H Toderici and Mark Stamp. Chi-squared distance and metamorphic virus detection. *Journal of Computer Virology and Hacking Techniques*, 2013.

[166] Hanchuan Peng, Fuhui Long, and Chris Ding. Feature selection based on mutual information criteria of max-dependency, max-relevance, and min-redundancy. *IEEE Transactions on pattern analysis and machine intelligence*, 2005.

[167] Mutual information. `http://www.scholarpedia.org/article/Mutual_information`. Accessed: 2018.

[168] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, 2011.

[169] Python Sklearn. `http://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.mutual_info_classif.html`. Accessed: 2018.

[170] Sung-Hyuk Cha. Comprehensive survey on distance/similarity measures between probability density functions. *City*, 2007.

[171] Introduction to K-means Clustering. `https://www.datascience.com/blog/k-means-clustering`. Accessed: 2018.

[172] Jiawei Han, Jian Pei, and Micheline Kamber. *Data mining: concepts and techniques*. Elsevier, 2011.

[173] Trupti M Kodinariya and Prashant R Makwana. Review on determining number of Cluster in K-Means Clustering. *International Journal*, 2013.

[174] Evgenia Dimitriadou, Sara Dolničar, and Andreas Weingessel. An examination of indexes for determining the number of clusters in binary data sets. *Psychometrika*, 2002.

[175] Xiaokui Shu, Danfeng Yao, and Naren Ramakrishnan. Unearthing stealthy program attacks buried in extremely long execution paths. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 401–413. ACM, 2015.

[176] Tielei Wang, Tao Wei, Zhiqiang Lin, and Wei Zou. IntScope: Automatically Detecting Integer Overflow Vulnerability in X86 Binary Using Symbolic Execution. In *NDSS*. Citeseer, 2009.

[177] MOV and MVN. `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0204j/Cihcdbca.html`. Accessed: 2018.

[178] Instruction substitution. `http://infocenter.arm.com/help/topic/com.arm.doc.dui0204j/Cihbeage.html#Cihgebhg`. Accessed: 2018.

[179] ARM Instruction Reference. `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0068b/CIHEDHIF.html`. Accessed: 2018.

[180] Hugo Gascon, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. Structural detection of android malware using embedded call graphs. In *Proceedings of the 2013 ACM workshop on Artificial intelligence and security*, pages 45–54. ACM, 2013.

[181] Shohei Hido and Hisashi Kashima. A linear-time graph kernel. In *Data Mining, 2009. ICDM'09. Ninth IEEE International Conference on*, pages 179–188. IEEE, 2009.

[182] Jonathan Oliver, Chun Cheng, and Yanggui Chen. TLSH–A Locality Sensitive Hash. In *Cybercrime and Trustworthy Computing Workshop (CTC), 2013 Fourth*, pages 7–13. IEEE, 2013.

[183] tlsh. `https://github.com/trendmicro/tlsh`. Accessed: 2018.

[184] Thomas Dullien and Rolf Rolles. Graph-based comparison of executable objects (english version). *SSTIC*, 5:1–3, 2005.

[185] Lannan Luo, Jiang Ming, Dinghao Wu, Peng Liu, and Sencun Zhu. Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection. In *ACM SIGSOFT*, 2014.

[186] He Huang. *Binary code reuse detection for reverse engineering and malware analysis*. PhD thesis, Concordia University, 2015.

[187] Sergey Ioffe. Improved consistent sampling, weighted minhash and l1 sketching. In *ICDM*, 2010.

[188] Apache Cassandra. `http://cassandra.apache.org/`. Accessed: 2018.

[189] Couchbase. `https://www.couchbase.com/`. Accessed: 2018.

[190] HBase. `https://hbase.apache.org/`. Accessed: 2018.

[191] MongoDB. `https://www.mongodb.com/`. Accessed: 2018.

[192] Benchmarking Top NoSQL Databases. `https://www.datastax.com/wp-content/themes/datastax-2014-08/files/NoSQL_Benchmarks_EndPoint.pdf?2`. Accessed: 2018.

[193] Docker. `https://www.docker.com/`. Accessed: 2018.

[194] Python Vs C++. `http://benchmarksgame.alioth.debian.org/u64q/compare.php?lang=python3&lang2=gpp`. Accessed: 2018.

[195] Pistache REST Framework. `https://github.com/oktal/pistache`. Accessed: 2018.

[196] cpprestsdk. `https://github.com/Microsoft/cpprestsdk`. Accessed: 2018.

[197] restbed. `https://github.com/Corvusoft/restbed`. Accessed: 2018.

[198] Pistache REST Framework Performance. `https://blog.binaryspaceship.com/2017/cpp-rest-api-frameworks-benchmark/#C_pistache`. Accessed: 2018.

[199] jsoncpp. `https://github.com/open-source-parsers/jsoncpp`. Accessed: 2018.

[200] boost. `http://www.boost.org/`. Accessed: 2018.

[201] boost-multiprecision. `http://www.boost.org/doc/libs/1_66_0/libs/multiprecision/doc/html/index.html`. Accessed: 2018.

[202] boost-graph. `http://www.boost.org/doc/libs/1_66_0/libs/graph/doc/index.html`. Accessed: 2018.

[203] logog. https://github.com/johnwbyrd/logog. Accessed: 2018.

[204] Datastax CPP-Driver. https://github.com/datastax/cpp-driver. Accessed: 2018.

[205] C++ LRU Cache. https://github.com/mohaps/lrucache11. Accessed: 2018.

[206] Docker-Compose. https://docs.docker.com/compose/.

[207] Python. https://www.python.org/. Accessed: 2018.

[208] Vagrant. https://www.vagrantup.com/. Accessed: 2018.

[209] David; GTM Research J. Leeds. The Networked Grid 150: The End-to-End Smart Grid Vendor Ecosystem Report and Rankings. 2012.

[210] David; GTM Research Groarke. The Networked Grid 150: The End-to-End Smart Grid Vendor Ecosystem Report and Rankings 2013. 2013.

[211] Greg Neichin, David Cheng, Sheeraz Haji, Josh Gould, Debjit Mukerji, and David Hague. 2010 US Smart Grid Vendor Ecosystem. 2010.

[212] Net-SNMP. http://www.net-snmp.org/. Accessed: 2018.

[213] NTP. http://www.ntp.org/. Accessed: 2018.

[214] cve-search-docker. https://github.com/leojcollard/cve-search-docker. Accessed: 2018.

[215] Napalm FTP-Indexer. https://www.searchftps.net/. Accessed: 2018.

[216] Cve Search. https://github.com/cve-search/cve-search. Accessed: 2018.

[217] ReadyNAS Firmware Image v6.1.6. http://www.downloads.netgear.com/files/GDC/READYNAS-100/ReadyNASOS-6.1.6-arm.zip. Accessed: 2018.

[218] Kai Chen, Peng Wang, Yeonjoon Lee, XiaoFeng Wang, Nan Zhang, Heqing Huang, Wei Zou, and Peng Liu. Finding unknown malice in 10 seconds: Mass vetting for new threats at the google-play scale. In *USENIX Security Symposium*, volume 15, 2015.