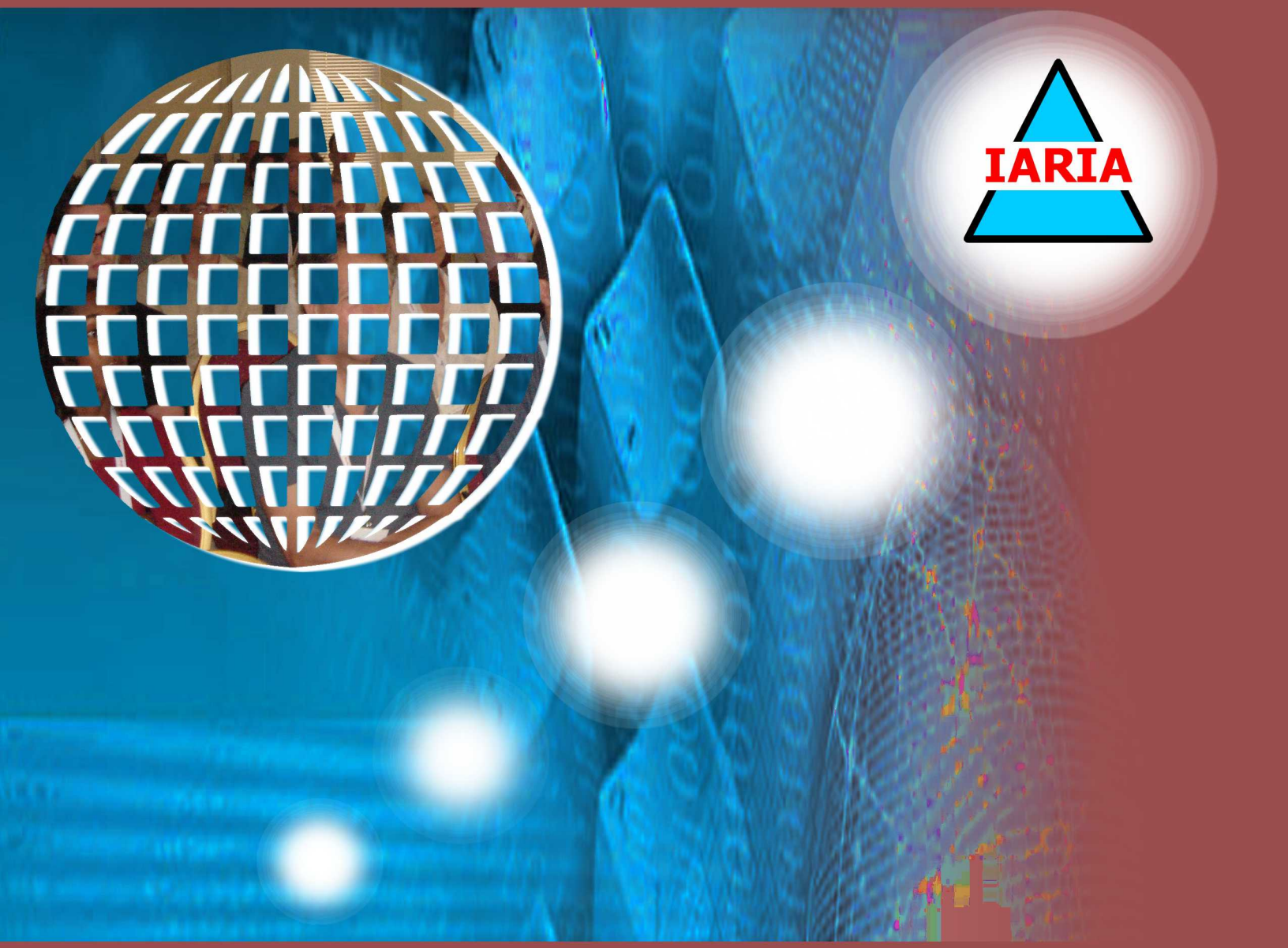
International Journal on Advances in Security



2022 vol. 15 nr. 1&2

The *International Journal on Advances in Security* is published by IARIA.
ISSN: 1942-2636
journals site: http://www.iariajournals.org
contact: petre@iaria.org

Sponsored by IARIA
www.iaria.org

International Journal on Advances in Security
Volume 15, Number 1 & 2, 2022

Sylvia Kierkegaard, IAITL-International Association of IT Lawyers, Denmark
Hyunsung Kim, Kyungil University, Korea
Geir M. Køien, University of Agder, Norway
Ah-Lian Kor, Leeds Metropolitan University, UK
Evangelos Kranakis, Carleton University - Ottawa, Canada
Lam-for Kwok, City University of Hong Kong, Hong Kong
Jean-Francois Lalande, ENSI de Bourges, France
Gyungho Lee, Korea University, South Korea
Clement Leung, Hong Kong Baptist University, Kowloon, Hong Kong
Diego Liberati, Italian National Research Council, Italy
Giovanni Livraga, Università degli Studi di Milano, Italy
Gui Lu Long, Tsinghua University, China
Jia-Ning Luo, Ming Chuan University, Taiwan
Thomas Margoni, University of Western Ontario, Canada
Rivalino Matias Jr ., Federal University of Uberlandia, Brazil
Manuel Mazzara, UNU-IIST, Macau / Newcastle University, UK
Catherine Meadows, Naval Research Laboratory - Washington DC, USA
Carla Merkle Westphall, Federal University of Santa Catarina (UFSC), Brazil
Ajaz H. Mir, National Institute of Technology, Srinagar, India
Jose Manuel Moya, Technical University of Madrid, Spain
Leonardo Mostarda, Middlesex University, UK
Jogesh K. Muppala, The Hong Kong University of Science and Technology, Hong Kong
Syed Naqvi, CETIC (Centre d'Excellence en Technologies de l'Information et de la Communication),Belgium
Sarmistha Neogy, Jadavpur University, India
Mats Neovius, Åbo Akademi University, Finland
Jason R.C. Nurse, University of Oxford, UK
Peter Parycek, Donau-Universität Krems, Austria
Konstantinos Patsakis, Rovira i Virgili University, Spain
João Paulo Barraca, University of Aveiro, Portugal
Sergio Pozo Hidalgo, University of Seville, Spain
Yong Man Ro, KAIST (Korea advanced Institute of Science and Technology), Korea
Rodrigo Roman Castro, University of Malaga, Spain
Heiko Roßnagel, Fraunhofer Institute for Industrial Engineering IAO, Germany
Claus-Peter Rückemann, Leibniz Universität Hannover / Westfälische Wilhelms-Universität Münster / North-German Supercomputing Alliance, Germany
Antonio Ruiz Martinez, University of Murcia, Spain
Paul Sant, University of Bedfordshire, UK
Peter Schartner, University of Klagenfurt, Austria
Alireza Shameli Sendi, Ecole Polytechnique de Montreal, Canada
Dimitrios Serpanos, Univ. of Patras and ISI/RC ATHENA, Greece
Pedro Sousa, University of Minho, Portugal
George Spanoudakis, City University London, UK
Vladimir Stantchev, Institute of Information Systems, SRH University Berlin, Germany
Lars Strand, Nofas, Norway
Young-Joo Suh, Pohang University of Science and Technology (POSTECH), Korea
Jani Suomalainen, VTT Technical Research Centre of Finland, Finland

Enrico Thomae, Ruhr-University Bochum, Germany
Tony Thomas, Indian Institute of Information Technology and Management - Kerala, India
Panagiotis Trimintzios, ENISA, EU
Peter Tröger, Hasso Plattner Institute, University of Potsdam, Germany
Simon Tsang, Applied Communication Sciences, USA
Marco Vallini, Politecnico di Torino, Italy
Bruno Vavala, Carnegie Mellon University, USA
Mthulisi Velempini, North-West University, South Africa
Miroslav Velev, Aries Design Automation, USA
Salvador E. Venegas-Andraca, Tecnológico de Monterrey / Texia, SA de CV, Mexico
Szu-Chi Wang, National Cheng Kung University, Tainan City, Taiwan R.O.C.
Steffen Wendzel, Fraunhofer FKIE, Bonn, Germany
Piyi Yang, University of Shanghai for Science and Technology, P. R. China
Rong Yang, Western Kentucky University , USA
Hee Yong Youn, Sungkyunkwan University, Korea
Bruno Bogaz Zarpelao, State University of Londrina (UEL), Brazil
Wenbing Zhao, Cleveland State University, USA

International Journal on Advances in Security
Volume 15, Numbers 1 & 2, 2022

CONTENTS

Improving Firewall Evolvability with an Iterated Local Search Algorithm

Geert Haerens

Department of Management Information Systems
Faculty of Business and Economics
University of Antwerp, Belgium
and Engie IT — Dir. Digital & Consulting
Email: geert.haerens@engie.be

Herwig Mannaert

Department of Management Information Systems
Faculty of Business and Economics
University of Antwerp, Belgium
Email: herwig.mannaert@uantwerp.be

***Abstract*—The Transmission Control Protocol/Internet Protocol (TCP/IP) based firewall is a notorious non-evolvable system. Changes to the firewall often result in unforeseen side effects, resulting in the unavailability of network resources. The root cause of these issues lies in the order sensitivity of the rule base and hidden relationships between rules. It is not only essential to define the correct rule. The rule must be placed at the right location in the rule base. As the rule base becomes more extensive, the problem increases. According to Normalized Systems, this is a Combinatorial Effect. In previous research, an artifact has been proposed to build a rule base from scratch in such a way that the rules will be disjoint from each other. Having disjoint rules is the necessary condition to eliminate the order sensitivity and thus the evolvability issues. In this paper, an algorithm, based on the Iterated Local Search metaheuristic, will be presented that will disentangle the service component in an existing rule base into disjoint service definitions. Such disentanglement is a necessary condition to transform a non-disjoint rule base into a disjoint rule base. The math behind the algorithm is presented, a demonstration using multiple firewall exports from a real operational environment is provided and the implications of the artifact are discussed.**

***Index Terms*—Firewall; Rule Base; Evolvability; Metaheuristic; Iterated Local Search.**

I. Introduction

This paper is an extended version of [1], and applies the researches discussed in [1] to operational firewall exports provided by Engie (an utilities multinational at which one of the author works). This paper also provides additional insights in the implications of using the research in an operational environment, as discussed in the PhD dissertation of one of the authors [2].

The TCP/IP based firewall has been and will continue to be an essential network security component in protecting network-connected resources from unwanted traffic. The increasing size of corporate networks and connectivity needs has resulted in firewall rule bases increasing considerably. Large rule bases have a nasty side effect. It becomes increasingly difficult to add the right rule at the correct location in the firewall. Anomalies start appearing in the rule base, resulting in the erosion of the firewall's security policy or incorrect functioning. Making changes to the firewall rule base becomes more complex as the size of the system grows. An observation shared by Forrester [3] and the firewall security industry [4] [5]. A more detailed literature review on the topic can be found in [6].

Normalized Systems (NS) theory [7] defines a Combinatorial Effect (CE) as the effect that occurs when the impact of a change is proportional to the nature of the change and the system's size. According to NS theory, a system that suffers from CE is considered unstable under change or non evolvable. A firewall suffers from CE. The evolvability issues are the root cause of the growing complexity of the firewall as time goes by.

The order sensitivity plays a vital role in the evolvability issues of the rule base. The necessary condition to remove the order sensitivity is known, being non-overlapping or disjoint rules. However, firewall rule bases do not enforce that condition, leaving the door open for misconfiguration. While previous work investigates the causes of anomalies [8] [9], detecting anomalies [10] [11] [12] and correcting anomalies at the time of entering new rules in the rule base [10], to the best of our knowledge and efforts, no work was found that tries to construct a rule base with ex-ante proven evolvability (= free of CE). While previous methods are reactive, this work proposes a proactive approach.

Issues with evolvability of the firewall rule base induce business risks. The first is the risk of technical communication paths not being available to execute business activities properly. The second is that flaws in the rule base may result in security issues, making the business vulnerable for malicious hacks resulting in business activities' impediment.

In this paper, we propose an artifact, an algorithm, that aims at converting an existing non-evolvable rule base into an evolvable rule base. Design Science [13] [14] is suited for research that wants to improve things through artifacts (tools, methods, algorithms, etc.). The Design Science Framework (see Figure 1) defines a relevance cycle (solve a real and relevant problem) and rigor cycle (grounded approach, usage of existing knowledge and methodologies).

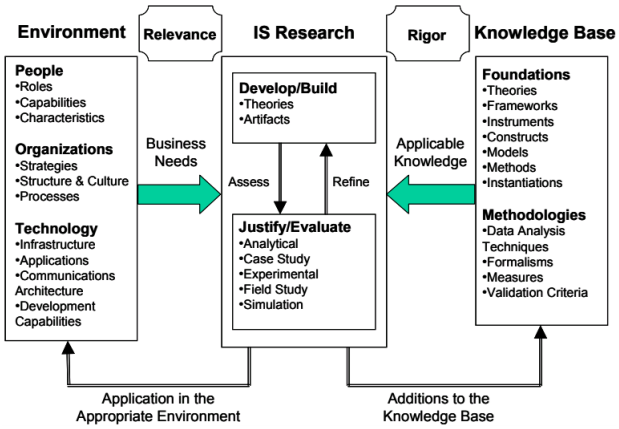


Fig. 1. The Design Science Framework - from [13] .

The Design Science Process (see Figure 2) guides the artifact creation process according to the relevance and rigor cycle. What follows is structured according to the Design Science process.

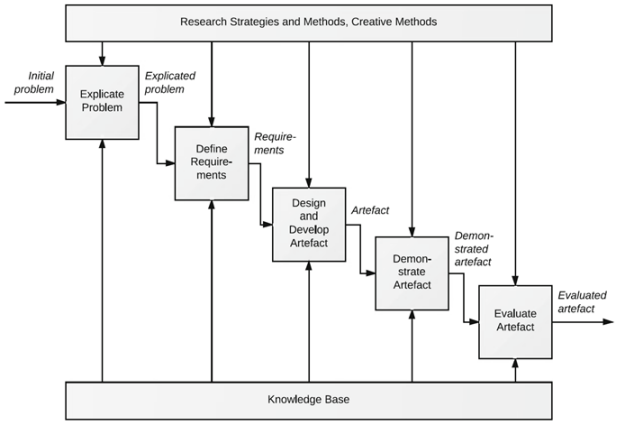


Fig. 2. The Design Science Process - from [14].

Section II introduces the basic concepts of firewalls, firewall rule relationships, Normalized Systems, and the evolvability issues of the firewall. In Section III, we will discuss the requirements for an algorithm that will transform a non-evolvable rule base, into an evolvable rule base. Section IV will build the different components of the proposed algorithm using the Iterated Local Search metaheuristic. In Section V, the algorithm will be demonstrated with real operational data. In Section VI, we evaluate and discuss our findings and we wrap-up with a conclusion in Section VII.

II. Problem Description

The first part of this section will explain how a firewall works and the concept of firewall group objects. The second part will discuss the relationships between firewall rules and introduces the Normalized Systems theory.

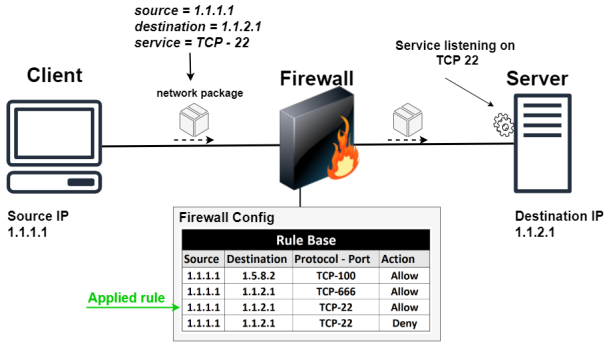


Fig. 3. Firewall concepts.

A. Firewall basics

An Internet Protocol Version 4 (IP4) TCP/IP based firewall, located in the network path between resources, can filter traffic between the resources, based on the Layer 3 (IP address) and Layer 4 (TCP/UDP ports) properties of those resources [15]. UDP stands for User Datagram Protocol and is, next to TCP, a post based communication protocol at the 4th level of the Open Systems Interconnection Model (OSI Model) [16]. Filtering happens by making use of rules. A rule is a tuple containing the following elements: <Source IP, Destination IP, Protocol, Destination Port, Action>. IP stands for IP address and is a 32-bit number that uniquely identifies a networked resource on a TCP/IP based network. The protocol can be TCP or UDP. Port is a 16-bit number (0 - 65.535) representing the TCP or UDP port on which a service is listening on the 4th layer of the OSI-stack.

When a firewall sees traffic coming from a resource with IP address =<Source IP>, going to resource =<Destination IP>, addressing a service listening on Port = <Destination port>, using Protocol = <Protocol>, the firewall will look for the first rule in the rule base that matches Source IP, Destination IP, Protocol and Destination Port, and will perform an action = <Action>, as described in the matched rule. The action can be "Allow" or "Deny". See Figure 3 for a graphical representation of the explained concepts. A firewall rule base is a collection of order-sensitive rules. The firewall starts at the top of the rule base until it encounters the first rule that matches the traffic. In a firewall rule, <Source IP>, <Destination IP>, <Destination Port> and <Protocol> can be one value or a range of values. In the remainder of this paper, protocol and port are grouped together in service (for example, TCP port 58 or UDP port 58 are 2 different services).

The firewalls discussed in this work are stateful, meaning that filtering happens on inbound traffic (towards the destination), but that the same firewall does not require rules to allow the response from the destination to the source. The firewall keeps track of the allowed inbound traffic and by default allows the response toward the source. In a stateless firewall this is not the case. A more elaborate discussion about the impact of inbound and outbound traffic on the evolvability of the firewall can be found in [2].

B. Firewall group objects

Rules containing IP addresses for source/destination and port numbers, are difficult to interpret by humans. Modern firewalls allow the usage of firewall objects, called groups, to give a logical name to a source, a destination, or a port, which is more human-friendly. Groups are populated with IP addresses or ports and can be nested. The groups are used in the definition of the rules. Using groups should improve the manageability of the firewall. See Figure 4 for an example.

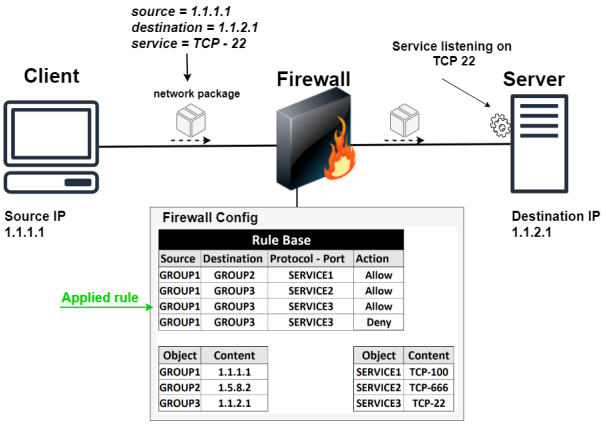


Fig. 4. Firewall concepts, including groups.

C. Firewall rule relationships

Based on [11], the relationships between rules and rule components are defined as follows:

- **Field**: A field in a rule is defined as a source, destination or service. A field is a set of values, with a minimum of size one.
  Example: *The source field of a rule contains 3 IP addresses/values - (10.10.10.1, 10.10.10.2, 10.10.10.3)*
- **Equal Fields**: Two corresponding fields of two rules are equal if the set of values of the fields are the same.
  Example: *The source field of a rule **R1** and source field in rule **R2** contain the same 3 IP addresses - (10.10.10.1, 10.10.10.2, 10.10.10.3)*
- **Inclusive Fields:** Two corresponding fields of two rules are inclusive if the set of values of the field of the first rule are a subset of, but not equal to, the second rule field's set of values.
  Example: *The source field of **R1** contains (10.10.10.1, 10.10.10.2) and the source field of **R2** contains (10.10.10.1, 10.10.10.2, 10.10.10.3). The IPs (10.10.10.1, 10.10.10.2) are a subset of (10.10.10.1, 10.10.10.2, 10.10.10.3). The source field of **R1** is inclusive with regards to the source field of **R2**.*
- **Correlated Fields**: Two corresponding fields of two rules are correlated if there are some values, but not all, of the field of the first rules that are equal to some values, but not all, of the field of the second rule. The intersection between the sets of values of the fields is not empty, but the fields are not equal or inclusive either.
  Example: *The source field of **R1** contains (10.10.10.1, 10.10.10.2, 10.10.10.3) and the source field of **R2** contains (10.10.10.2, 20.20.20.20, 30.30.30.30). The two source fields are correlated as they intersect with the IP 10.10.10.2.*
- **Distinct Fields:** Two corresponding fields in two rule are distinct if they are not equal, not inclusive or not correlated. The intersection between the sets of values of the fields is empty.
  Example: *Source field (10.10.10.10) of rule **R1** and source field (10.10.10.100) of rule **R2** are distinct.*
- **Matching Fields:** Two corresponding fields in two rules match if they are equal or inclusive.
  Example: *Source field of **R1** = (10.10.10.1, 10.10.10.10) and the source field of **R2** = (10.10.10.1, 10.10.10.10, 10.10.10.30), are matching.*
- **Exactly Matching Rules:** Rules **R1** and **R2** are exactly matched if every field in **R1** is equal to the corresponding field in **R2**.
  Example: *Rule **R1**: (source = (10.10.10.10); destination = (20.20.20.20); service = (TPC 100); action = allow) and **R2**: (source = (10.10.10.10); destination = (20.20.20.20); service = (TPC 100); action = deny), are exactly matching rules.*
- **Completely Disjoint Rules:** Rules **R1** and **R2** are completely disjoint if every field in **R1** and **R2** is distinct.
  Example: *Consider rule **R1**: (source = (10.10.10.10); destination = (20.20.20.20); service = (TPC 100); action = allow) and rule **R2**: (source = (30.30.30.30, 30.30.30.21); destination = (40.40.40.40, 40.40.40.41); service = (TPC 200,201); action = deny). Both rules are completely disjoint.*
- **Partially Disjoint Rules or Partially Matching Rules:** Rules **R1** and **R2** are partially disjoint (or partially matched) if there is at least one field in **R1** and **R2** that is distinct. The other fields can be equal, inclusive or correlated.
  Example: *Consider rule **R1**: (source = (10.10.10.10); destination = (20.20.20.20); service = (TPC 100); action = allow) and rule **R2**: (source = (10.10.10.10); destination = (40.40.40.40, 40.40.40.41); service = (TPC 100,201); action = deny). **R1** and **R2** are partially disjoint, as destination is a distinct field.*
- **Inclusively Matching Rules:** Rules **R1** and **R2** are inclusively matched if there is at least one field that is inclusive, and the remaining fields are either inclusive or equal.
  Example: *Consider Rule **R1**: (source = (10.10.10.10); destination = (20.20.20.20); service = (TPC 100); action = allow) and **R2**: (source = (10.10.10.10, 10.10.10.11); destination = (20.20.20.20, 20.20.20.21); service = (TPC 100); action = deny). Then rule **R1** inclusively matches rule **R2**.*
- **Correlated Rules**: Rules **R1** and **R2** are correlated there is at least one field that is correlated, while the remaining fields are either equal or inclusive.

***Example**: Consider rule **R1**: source = (10.10.10.10, 10.10.10.11); destination = (20.20.20.20, 40.40.40.41); service = (TPC 100); action = allow) and rule **R2**: (source = (10.10.10.10); destination = (40.40.40.40, 40.40.40.41); service = (TPC 100,201); action = deny). Rules **R1** and **R2** are correlated.*

Figure 5 represents the different relations in a graphical manner. Exactly matching, inclusively matching and correlated rules can result in the following firewall anomalies [10]:

- *Shadowing Anomaly*: A rule **Rx** is shadowed by another rule **Ry** if **Ry** precedes **Rx** in the policy, and **Ry** can match all the packets matched by **Rx**. The result is that **Rx** is never activated.
- *Correlation Anomaly*: Two rules **Rx** and **Ry** can cause a correlation anomaly if, the rules **Rx** and **Ry** are correlated and if **Rx** and **Ry** have different filtering actions.
- *Redundancy Anomaly*: A redundant rule **Rx** performs the same action on the same packets as another rule **Ry** so that if **Rx** is removed the security policy will not be affected.

A fully consistent rule base should only contain disjoint (completely or partial) rules. In that case, the order of the rules in the rule base is of no importance, and the anomalies described above will not occur [8] [9] [10] . However, due to several reasons such as unclear requirements, a faulty change management process, lack of organization, manual interventions, and system complexity [13], the rule base will include correlated, exactly matching, and inclusively matching rules, and thus resulting in evolvability issues.

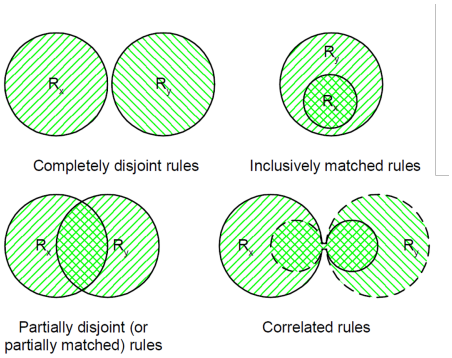


Fig. 5. Possible relationships between rules (from [11]).

D. Normalized Systems concepts

Normalized Systems (NS) theory [7] [17] originates from the field of software development. NS theory takes the concept of system theoretic stability from the domain of classic engineering to determine the necessary conditions a modular structure of a system must adhere to in order for the system to exhibit stability under change. Stability is defined as Bounded Input results in Bounded Output (BIBO). Transferring this concept to software design, one can consider bounded input as a certain amount of functional changes to the software and the bounded output as the number of effective software changes. If the amount of effective software changes is not only proportional to the amount of functional changes but also the size of the existing software system, then NS theory states that the system exhibits a CE and is considered unstable under change.

NS theory proves that, in order to eliminate CE, the software system must have a certain modular structure, were each module respects four design rules. Those rules are:

- Separation of Concern (SoC): a module should only address one concern or change driver.
- Separation of State (SoS): a state should separate the use of a module by another module during its operation.
- Action Version Transparency (AVT): a module, performing an action should be changeable without impacting modules calling this action.
- Data Version Transparency (DVT): a module performing a certain action on a data structure, should be able to continue doing this action, even is the data structures has undergone change (add/remove attributes).

NS theory can be used to study evolvability in any system, which can be seen as a modular system and derive design criteria for the evolvability of such a system [18] [19].

III. Requirements for the Solution

In [1] [2] [6] the necessary conditions for an evolvable firewall rule base are discussed. All the rules in the rule base must be disjoint or partially disjoint from each other. In [6] an artifact, a method, is proposed to create disjoint rules. Following the method will result in a firewall rule base that is free from CE for ADD and REMOVE changes.

For a given network **N**, containing $\mathbf{C_j}$ sources and $\mathbf{H_j}$ destinations, offering 2^{17} services (protocol/port) (= the max amount of possible UDP and TCP ports according to the TCP/IP V4 standard), and having a firewall **F** between the sources and the destinations, it can be shown (see [6]) that $\mathbf{f_{max}}$ is the number of possible rules (including both "allow" and "deny" rules) that can be defined on the firewall **F** (the solution space):

$$\mathbf{f_{max}} = 2.\left(\sum_{a=1}^{Hj}\binom{Cj}{a}\right).\left(\sum_{a=1}^{Hj}\binom{Hj}{a}\right).\left(\sum_{k=1}^{2^{17}}\binom{2^{17}}{k}\right) \quad (1)$$

where $\mathbf{C_j}$ and $\mathbf{H_j}$ are function of **N**: $\mathbf{C_j} = f_c(\mathbf{N})$ and $\mathbf{H_j} = f_h(\mathbf{N})$

Out of this design space, the amount of firewall rules that will exhibit ex-ante proven evolvability are the explicit "allow" rules that are disjoint, and it equal to:

$$\mathbf{f_{disjoint}} = \mathbf{H_j}.2^{17} \quad (2)$$

where $\mathbf{H_j}$ is the number of hosts connected to the network. $\boldsymbol{H_j} = \boldsymbol{fh(N)}$ and 2^{17} the max amount of services available on a host.

Applying the artifact drastically reduced the solution space. The artifact describes the green-field situation; building a rule

base from scratch. The luxury of a green-field is often not present. We require a solution that can convert an existing rule base, into a rule rule base that only contains disjoint rules. Of course, the original filtering strategy expressed in the rule base must stay the same. From [1] [2] [6] we know that we require disjoint service definitions. If we can disentangle the service definitions, and adjust the rules accordingly, we have our basic building block for a disjoint rule base. For each disjoint service definition, we need to create as many destination groups as there are host offering that service (lookup in rule base), and for each host-service combination, we require one source group definition. All components are then present to expand a non-evolvable rule base into a normalized evolvable rule base. Figure 6 visualizes what we want the solution to do.

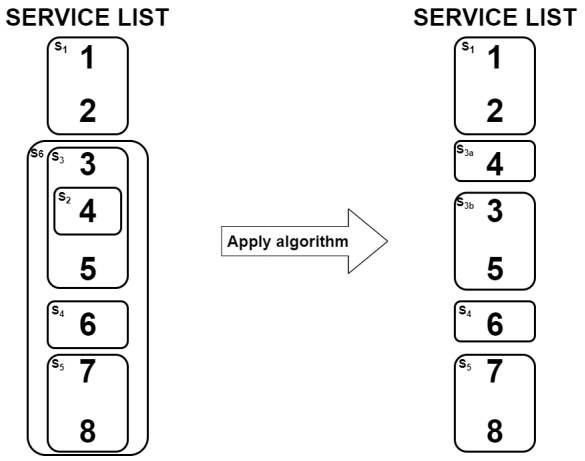


Fig. 6. Algorithm objective.

IV. Artifact Design

In this section we will discuss a different artifact, the brown-field artifact, that will convert a non evolvable rule base into an evolvable rule base, by disentangling the service definitions. The different components that comprise the algorithm will be discussed in dept. We begin by rationalizing the choice for Iterated Local Search (ILS) as metaheuristic [20] [21] [22]. We will discuss the nature of the initial solution, the set of feasible solutions, and the objective function associated with a solution. We continue by defining the move type, move strategy, perturbation and stop condition of the Iterated Local Search. The final part of this section provides a high level algorithm that represents the brown-field artifact.

A. Metaheuristic Selection

The objective is to disentangle/reshuffle the service definitions into a set of new service definitions that are disjoint but maximally large. The simplest solution is to create one service definition per port. However, some ports belong together to deliver a service. This filtering logic is embedded in the rule base and service definitions. It must be preserved.

Service definitions containing ports that appear in multiple service definitions must be split into non-overlapping service definitions. The result should be that the degree of overlap (or disjointness) of all service definitions decreases as more service definitions are split.

Let us say that a user measures the degree of disjointness of the entirety of the service definitions (pre-change and post-change) and then observes a post-change improvement in the degree of disjointness. It would be correct to conclude that the change represents an improvement to the previous version.

A Local Search (LS) heuristic is a suitable method for organizing such gradual improvement processes. To avoid getting stuck in a local optimum (see further), the Local Search will be upgraded to an Iterated Local Search. The Iteration component should result in avoiding becoming stuck in a local optimum where we can no longer perform splits and improve the disjointness. The Iteration component should perform a special kind of split called a ”perturbation” that will allow the continuation of the search for improvement.

B. Initial Solution and Neighborhood

The initial solution is the rule base containing all of the service definitions. It is the rule base with all the service definitions. The set of all service definitions is our neighborhood. We will have to pick a service definition, confirm whether or not it is disjoint and, if not, split it and see how this affects the solution - that is whether or not disjointness has improved. The solution space (SP) for the service definitions consists of all possible combinations of ports. If the number of distinct ports in the service groups equals **P**, then the **SP** is:

$$\mathbf{SP} = \sum_{k=1}^{\mathbf{P}} \binom{\mathbf{P}}{k} \tag{3}$$

P can be max 2^{17}. We are looking to find a new solution that is part of the solution space, in which all service definitions are disjoint yet grouped within groups of maximum size.

C. Objective Function

To know whether or not the splitting of a service definition results in improving the solution, we need a mechanism to express the degree of disjointness of a service definition and of the total rule base.

Let $\boldsymbol{p}$ represent a service port.

Let **S** be a set of ports, representing a service definition.
$\mathbf{S} = \{ \boldsymbol{p}_1 ... \boldsymbol{p}_{nS} \}$
where$|\ \mathbf{S}\ | = nS$ = number of ports in the service definition.

Let σ be the set of all service definitions $\mathbf{S}_i$ used in the firewall rule base.
$\sigma = \{\mathbf{S}_1 ... \mathbf{S}_{n\sigma}\}$
where$|\ \sigma\ | = n\sigma$= number of service definitions

Let $\mathbf{P}\mathrm{F}(\boldsymbol{p}_x)_\sigma$ be the port frequency of port $\boldsymbol{p}_x$ in σ, as

the number of times $\boldsymbol{p}_x$ is used in services of σ.

$$\mathbf{PF}(\boldsymbol{p}_x)\sigma = \sum_{i=1}^{n\sigma} \mid \mathbf{S}_\mathrm{i} \cap \boldsymbol{p}_x \mid \tag{4}$$

We define the Disjointness Index $\mathbf{DI}(\mathbf{S}_\mathrm{x})_\sigma$, of a service definition $\mathbf{S}_\mathrm{x}$, in σ as the sum of the port frequencies $\mathbf{PF}(\boldsymbol{p}_x)_\sigma$ of the ports $\boldsymbol{p}x$ of $\mathbf{S}_\mathrm{x}$, divided by the number of ports in $\mathbf{S}_\mathrm{x}$.

$$\mathbf{DI}(S_\mathrm{x})\sigma = \frac{\sum_{i=1}^{nx} PF(p_x)_\sigma}{nx} \tag{5}$$

where nx = $\mid S_\mathrm{x} \mid$= number of ports in $\mathbf{S_x}$.

A disjoint service is a service whereby each port **p** appears in only one service definition. The **DI** of a disjoint service will have a value of **1** and a value greater then 1 if the service is not disjoint.

We define the Objective Function $\mathbf{OF}_\sigma$, in σ, as the sum of all $\mathbf{DI}(\mathbf{S}_\mathrm{x})_\sigma$ and of all service definitions in σ.

$$\mathbf{OF}\sigma = \sum_{i=1}^{n\sigma} \mathbf{DI}(\mathbf{S}_i)\sigma \tag{6}$$

with $n\sigma$ the number of service definitions in the solution.

We define an Optimal Solution as a solution where $\mathbf{OF}\sigma$ equals the number of service definitions, as this means that all **DI** of all service definitions are equal to 1.

$$\mathbf{OF}\sigma = \mid \sigma \mid \tag{7}$$

An Optimal Solution is not necessarily a Global Optimum as making service definitions of one port would also yield an objective function value that is equal to the total number of service definitions.

D. Feasible Solutions

Whatever kind of splits we will be performing, the original filtering logic of the rule base must be maintained. When a service is split, all rules that contain this service must to be modified. The original service must be replaced by the result of the split. As we want a rule to contain only one service definition, it may be required to split the rules containing the split result.

Example: R1 contains service Sx. Sx is split into Sx1 and Sx2. To reflect this, we replace Sx with Sx1 and Sx2 in rule R1. However, a rule must only contain one service. R1 needs to be split into R1.1 and R1.2, where R1.1 is a copy of R1 but with Sx being replaced by Sx1, and R1.2 is a copy of R1 but with Sx being replaced by Sx2. Both rules are put in consecutive locations in the rule base.

E. Move Type

Before we decide on the move type, we must first investigate the impact that splitting of service definitions has on the objective function. Based on this analysis, a selection of type of split (move type) will be made.

*1) The Impact of Splitting Service Definitions on the **OF**:*

A service definition can:

- be a subset of existing service definitions.
- be the superset of existing service definitions.
- be partially overlapped with other service definitions.
- be a combination of the above.

Let $\mathbf{S}_\mathrm{ca}$ be the candidate service we will split.

$$\begin{cases} \mathbf{S}_\mathrm{ca} = \{\boldsymbol{p}_1 ... \boldsymbol{p}_{nca}\} \\ nca = \mid \mathbf{S}_\mathrm{ca} \mid = \text{number of ports in the } \mathbf{S}_\mathrm{ca} \end{cases}$$

Let $\mathbf{S}_\mathrm{co}$ be an arbitrary set of ports that are part of $\mathbf{S}_\mathrm{ca}$, making up the new service $\mathbf{S}_\mathrm{co}$, that is to be extracted from $\mathbf{S}_\mathrm{ca}$.

$$\begin{cases} \mathbf{S}_\mathrm{co} = \{\boldsymbol{p}_j ... \boldsymbol{p}_{j+nco}\} \\ nco = \mid \mathbf{S}_\mathrm{co} \mid = \text{number of ports in the } \mathbf{S}_\mathrm{co}. \\ \mathbf{S}_\mathrm{ca} \cap \mathbf{S}_\mathrm{co} = \{\boldsymbol{p}_j ... \boldsymbol{p}_{j+nco}\} \\ \mid \mathbf{S}_\mathrm{ca} \cap \mathbf{S}_\mathrm{co} \mid = nco \end{cases}$$

Let $\mathbf{S'}_\mathrm{ca}$ be the new service comprised of ports that are part of $\mathbf{S}_\mathrm{ca}$ but not of $\mathbf{S}_\mathrm{co}$. $\mathbf{S'}_\mathrm{ca}$ is what is left of $\mathbf{S}_\mathrm{ca}$, after splitting-up or carving-out $\mathbf{S}_\mathrm{co}$

$$\begin{cases} \mathbf{S'}_\mathrm{ca} = \mathbf{S}_\mathrm{ca} \backslash \mathbf{S}_\mathrm{co} = \{\boldsymbol{p}_1 ... \boldsymbol{p}_{j-1}, \boldsymbol{p}_{j+nco+1}, ... \boldsymbol{p}_{nca}\} \\ \mid \mathbf{S'}_\mathrm{ca} \mid = nca - nco \end{cases}$$

Let $\sigma_{\mathbf{S}_{ca}}$ be the set of services that contains ports that are also part of service $\mathbf{S}_\mathrm{ca}$.

$$\begin{cases} \sigma_{\mathbf{S}_{ca}} = \{\mathbf{S}_{\mathrm{V}1} ... \mathbf{S}_{\mathrm{V}n}\} \\ \forall \mathbf{S}_{\mathrm{V}x} \ \mathrm{x}{=}1 \rightarrow \mathrm{n} \mid \\ * \ \mid \mathbf{S}_\mathrm{ca} \cap \mathbf{S}_{\mathrm{V}x} \mid \neq \emptyset \\ * \ \mid \mathbf{S}_{\mathrm{V}x} \mid = Vnx \\ * \ \mid \mathbf{S}_\mathrm{ca} \cap \mathbf{S}_{\mathrm{V}x} \mid = \mathrm{q_x} = \text{the amount of port overlap between } \mathbf{S}_\mathrm{ca} \text{ and } \mathbf{S}_{\mathrm{V}x} \end{cases}$$

See Figure 7 for a visual representation of these definitions.

When the split or carve-out of **S**co from **S**ca is performed, the port frequencies, the **DI** and the **OF** change, depending on the effect of the split. We shall now investigate under which conditions the split will improve the objective function.

Let σ_B be the set of services before the split and σ_A be the set of services after the split. We want to know which conditions will improve the Objective Function, or

$$\begin{cases} \Delta\mathbf{OF} = \mathbf{OF}_{\sigma_B} - \mathbf{OF}_{\sigma_A} > 0 \\ \Delta\mathbf{OF} > 0 \text{ means } \mathbf{OF} \text{ improved (=lowered).} \\ \Delta\mathbf{OF} < 0 \text{ means } \mathbf{OF} \text{ deteriorated (=increased).} \end{cases}$$

$\mathbf{S}_\mathrm{co}$ is a random subgroup of $\mathbf{S}_\mathrm{ca}$, meaning not necessarily part of σ_B. Subsequent to a split $\mathbf{S}_\mathrm{ca}$ becomes $\mathbf{S'}_\mathrm{ca}$. Both $\mathbf{S'}_\mathrm{ca}$ and $\mathbf{S}_\mathrm{co}$ are part of σ_A.

There are three possible cases:

- $\mathbf{S'}_\mathrm{ca}$ and $\mathbf{S}_\mathrm{co}$ also exist in σ_B. The split results in two existing services. We merge them into the existing services — the split results in a reduction of the total number of services with 1.
  $\mid \sigma_A \mid$ - $\mid \sigma_B \mid$ = -1

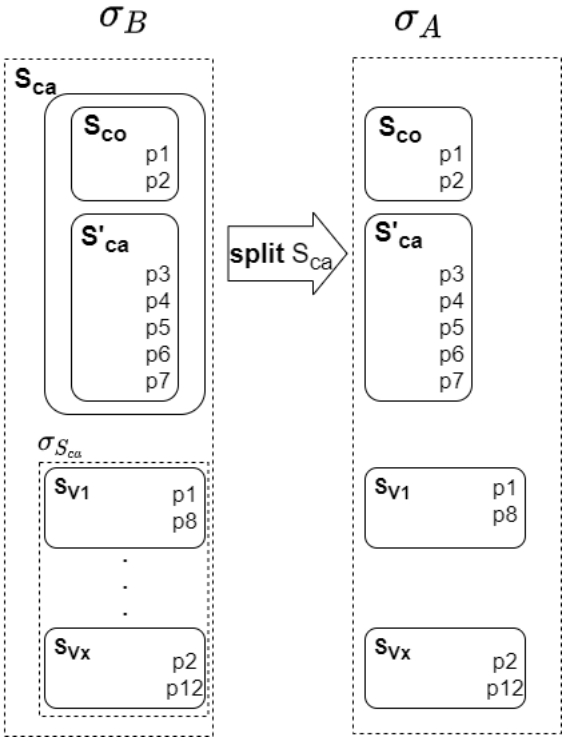


Fig. 7. Split example

- **S'**$_{ca}$ or $\mathbf{S}_{co}$ exists in σ_B. The split results in a new service and an existing service. The existing service merges and the split results in an equal number of services.
  $| \sigma_A | - | \sigma_B | = 0$
- **S'**$_{ca}$ and $\mathbf{S}_{co}$ do not exist in σ_B. The split results in two new services and the split results in an increase of the total number of services with 1.
  $| \sigma_A | - | \sigma_B | = +1$

We shall now investigate what kind of change in Objective Function value we can expect, based on the three following cases.

Case 1: $| \sigma_B | - | \sigma_A | = -1$
S'$_{ca}$ and $\mathbf{S}_{co}$ are elements of σ_A and σ_B.
$\mathbf{S}_{ca}$ only exists in σ_B.
$\mathbf{S}_{ca} = \{p_1 ... p_{nca}\}$
As $\mathbf{S}_{ca}$ is not part of σ_A, the port frequencies of all ports of S_{ca} decreased by 1 in σ_A.

$$\forall \mathbf{p} \in \mathbf{S}_{ca} \mid (\mathbf{PF}(\mathbf{p})_{\sigma_A} = \mathbf{PF}(\mathbf{p})_{\sigma_B} - 1 \tag{8}$$

See Figure 8 for a graphical representation.

As the port frequencies of all ports that are part of $\mathbf{S}_{ca}$ decrease, the **DI** of all groups that contain one or more port of $\mathbf{S}_{ca}$ are also impacted. These are all $\mathbf{S}_{Vi}$ service groups.

When calculating ΔOF, only the impacted service groups must be taken into account.

$$\Delta\mathbf{OF} = \mathbf{OF}_{\sigma_B} - \mathbf{OF}_{\sigma_A}$$

$$\implies \Delta\mathbf{OF} = [\mathbf{DI}(\mathbf{S}_{ca})_{\sigma_B} + \mathbf{DI}(\mathbf{S'}_{ca})_{\sigma_B} + \mathbf{DI}(\mathbf{S}_{co})_{\sigma_B} + \sum_{i=1}^{n} \mathbf{DI}(\mathbf{S}_{Vi})_{\sigma_B}] - [\mathbf{DI}(\mathbf{S'}_{ca})_{\sigma_A} + \mathbf{DI}(\mathbf{S}_{co})_{\sigma_A} + \sum_{i=1}^{n} \mathbf{DI}(\mathbf{S}_{Vi})_{\sigma_A}]$$

$$\implies \Delta\mathbf{OF} = \mathbf{DI}(\mathbf{S}_{ca})_{\sigma_B} + [\mathbf{DI}(\mathbf{S'}_{ca})_{\sigma_B} - \mathbf{DI}(\mathbf{S'}_{ca})_{\sigma_A}] + [\mathbf{DI}(\mathbf{S}_{co})_{\sigma_B} - \mathbf{DI}(\mathbf{S}_{co})_{\sigma_A}] + [\sum_{i=1}^{n} \mathbf{DI}(\mathbf{S}_{Vi})_{\sigma_B} - \sum_{i=1}^{n} \mathbf{DI}(\mathbf{S}_{Vi})_{\sigma_A}]$$

Taking (5) and (8) into account:

(a) $\mathbf{DI}(\mathbf{S'}_{ca})_{\sigma_A} = \frac{\sum_{i=1}^{nca-nco} \mathbf{PF}(\mathbf{p}_i)_{\sigma_A}}{nca-nco}$

$$\implies \mathbf{DI}(\mathbf{S'}_{ca})_{\sigma_A} = \frac{\sum_{i=1}^{nca-nco} \mathbf{PF}(\mathbf{p}_i)_{\sigma_B} - (nca-nco)}{nca-nco}$$

$$\implies \mathbf{DI}(\mathbf{S'}_{ca})_{\sigma_A} = \mathbf{DI}(\mathbf{S'}_{ca})_{\sigma_B} - 1$$

$$\implies \mathbf{DI}(\mathbf{S'}_{ca})_{\sigma_B} - \mathbf{DI}(\mathbf{S'}_{ca})_{\sigma_A} = 1$$

(b) $\mathbf{DI}(\mathbf{S}_{co})_{\sigma_A} = \frac{\sum_{i=1}^{nco} \mathbf{PF}(\mathbf{p}_i)_{\sigma_A}}{nco}$

$$\implies \mathbf{DI}(\mathbf{S}_{co})_{\sigma_A} = \frac{\sum_{i=1}^{nco} \mathbf{PF}(\mathbf{p}_i)_{\sigma_B} - (nco)}{nco} = \mathbf{DI}(\mathbf{S}_{co})_{\sigma_B} - 1$$

$$\implies \mathbf{DI}(\mathbf{S}_{co})_{\sigma_B} - \mathbf{DI}(\mathbf{S}_{co})_{\sigma_A} = 1$$

(c) $\sum_{i=1}^{n} \mathbf{DI}(\mathbf{S}_{Vi})_{\sigma_B} - \sum_{i=1}^{n} \mathbf{DI}(\mathbf{S}_{Vi})_{\sigma_A} =$

$$\sum_{i=1}^{n} \frac{\sum_{j=1}^{nvi} \mathbf{PF}(\mathbf{p}_j)_{\sigma_B}}{nvi} - \sum_{i=1}^{n} \frac{\sum_{j=1}^{nvi} \mathbf{PF}(\mathbf{p}_j)_{\sigma_A}}{nvi} =$$

$$\sum_{i=1}^{n} \frac{\sum_{j=1}^{nvi} \mathbf{PF}(\mathbf{p}_j)_{\sigma_B}}{nvi} - \sum_{i=1}^{n} \frac{\sum_{j=1}^{nvi} \mathbf{PF}(\mathbf{p}_j)_{\sigma_B} - qvi}{nvi}$$

$$\implies \sum_{i=1}^{n} \mathbf{DI}(\mathbf{S}_{Vi})_{\sigma_B} - \sum_{i=1}^{n} \mathbf{DI}(\mathbf{S}_{Vi})_{\sigma_A} = \sum_{i=1}^{n} \frac{qvi}{nvi}$$

Putting (a), (b) and (c) into ΔOF

$$\Delta\mathbf{OF} = \mathbf{DI}(\mathbf{S}_{ca})_{\sigma_B} + 2 + \sum_{i=1}^{n} \frac{qvi}{nvi}$$

Conclusion: If $| \sigma_B | - | \sigma_A |$ ***= -1, then*** Δ***OF is always*** > 0 ***(all terms are positive), and the Objective Function always improves.***

Case 2: $| \sigma_B | - | \sigma_A | = 0$
S'$_{ca}$ or $\mathbf{S}_{co}$ are part of σ_A or σ_B (exclusive OR).
Assume $\mathbf{S}_{co}$ already exists in σ_B (carve-out of an existing group)
$\mathbf{S}_{ca}$ does not exist in σ_A, but **S'**$_{ca}$ does exist in σ_A. The port frequencies of all ports of **S'**$_{ca}$ do not change.

$$\forall \mathbf{p} \in \mathbf{S}_{ca} \setminus \mathbf{S}_{co} \mid \mathbf{PF}(\mathbf{p})_{\sigma_A} = \mathbf{PF}(\mathbf{p})_{\sigma_B} \tag{9}$$

$\mathbf{S}_{co}$ already exists in σ_B. The group cancels out in σ_B and the port frequencies of all ports in $\mathbf{S}_{co}$ decrease.

$$\forall \mathbf{p} \in \mathbf{S}_{co} \mid \mathbf{PF}(\mathbf{p})_{\sigma_A} = \mathbf{PF}(\mathbf{p})_{\sigma_B} - 1 \tag{10}$$

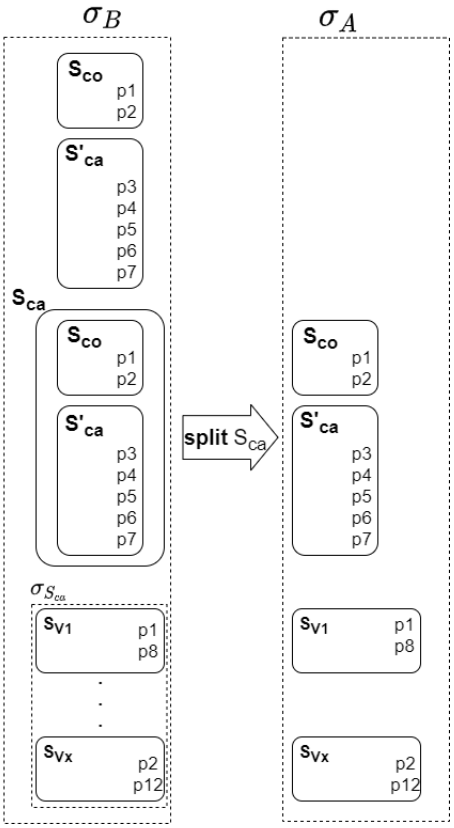


Fig. 8. Split case 1

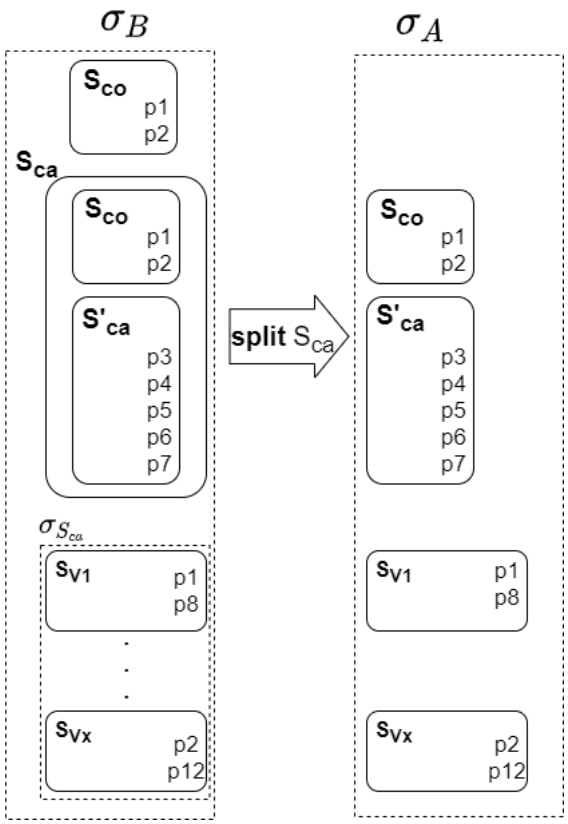


Fig. 9. Split case 2

See Figure 9 for a graphical representation.

As the port frequencies decrease, the **DI** of all groups that contain one or more port of $\mathbf{S}_{\text{co}}$ are also impacted. These are all $\mathbf{S}_{\text{Vi}}$ service groups.

When calculating Δ**OF**, only impacted service groups must be taken into account.

$$\Delta\mathbf{OF} = [\mathbf{DI}(\mathbf{S}_{ca})_{\sigma_B} + \mathbf{DI}(\mathbf{S}_{co})_{\sigma_B} + \sum_{i=1}^{n}\mathbf{DI}(\mathbf{S}_{Vi})_{\sigma_B}] - [\mathbf{DI}(\mathbf{S}_{ca})_{\sigma_A} + \mathbf{DI}(\mathbf{S}_{co})_{\sigma_A} + \sum_{i=1}^{n}\mathbf{DI}(\mathbf{S}_{Vi})_{\sigma_A}]$$

$$\Longrightarrow \Delta\mathbf{OF} = [\mathbf{DI}(\mathbf{S}_{ca})_{\sigma_B} - \mathbf{DI}(\mathbf{S'}_{ca})_{\sigma_A}] + [\mathbf{DI}(\mathbf{S}_{co})_{\sigma_B} - \mathbf{DI}(\mathbf{S}_{co})_{\sigma_A}] + [\sum_{i=1}^{n}\mathbf{DI}(\mathbf{S}_{Vi})_{\sigma_B} - \sum_{i=1}^{n}\mathbf{DI}(\mathbf{S}_{Vi})_{\sigma_A}]$$

Taking (5) and (10) into account and using the same type of calculations as in Case 1:

(d) $\mathbf{DI}(\mathbf{S}_{ca})_{\sigma_B} - \mathbf{DI}(\mathbf{S'}_{ca})_{\sigma_A} = \mathbf{DI}(\mathbf{S}_{ca})_{\sigma_B} - \mathbf{DI}(\mathbf{S'}_{ca})_{\sigma_B}$

(e) $\mathbf{DI}(\mathbf{S}_{co})_{\sigma_B} - \mathbf{DI}(\mathbf{S}_{co})_{\sigma_A} = 1$

(f) $\mathbf{DI}(\mathbf{S'}_{ca})_{\sigma_A} + \mathbf{DI}(\mathbf{S}_{co})_{\sigma_A} + \sum_{i=1}^{n}\mathbf{DI}(\mathbf{S}_{Vi})_{\sigma_A} = \sum_{i=1}^{n}\frac{qvi}{nvi}$ (see case 1)

Putting (d), (e) and (f) into ΔOF

$$\Delta\mathbf{OF} = \mathbf{DI}(\mathbf{S}_{ca})_{\sigma_B} - \mathbf{DI}(\mathbf{S'}_{ca})_{\sigma_B} + 1 + \sum_{i=1}^{n}\frac{qvi}{nvi} > 0$$

The same result is obtained when the assumption is made that $\mathbf{S'}_{\text{ca}}$ already exits in σ_B

Conclusion: If $|\ \sigma_B\ |$ **-** $|\ \sigma_A\ |$ **= 0, then** Δ***OF (due to*** $-\mathbf{DI}(\mathbf{S'}_{ca})_{\sigma_B}$***) can be*** < 0***, and the Objective Function can thus deteriorate***

Case 3: $|\ \sigma_B\ |$ **-** $|\ \sigma_A\ |$ **= 1**

$\mathbf{S}_{\text{ca}}$ splits into 2 mutually-exclusive new services ($\mathbf{S'}_{\text{ca}}$ and $\mathbf{S}_{\text{co}}$). Neither $\mathbf{S'}_{\text{ca}}$ nor $\mathbf{S}_{\text{co}}$ are part of σ_B.
$\mathbf{S'}_{\text{ca}}$ and $\mathbf{S}_{\text{co}}$ are both part of σ_A.
The port frequencies **PF** of any **p** do not change. No other services are impacted.

$$\forall \mathbf{p} \in \mathbf{S}_{ca} \mid \mathbf{PF}(\mathbf{p})_{\sigma_A} = \mathbf{PF}(\mathbf{p})_{\sigma_B} \tag{11}$$

The only factors playing a role in the calculation of Δ**OF** are $\mathbf{DI}(\mathbf{S}_{ca})_{\sigma_B}, \mathbf{DI}(\mathbf{S'}_{ca})_{\sigma_A}$, and $\mathbf{DI}(\mathbf{S}_{co})_{\sigma_A}$

$$\Delta\mathbf{OF} = \mathbf{DI}(\mathbf{S}_{ca})_{\sigma_B} - \mathbf{DI}(\mathbf{S'}_{ca})_{\sigma_A} - \mathbf{DI}(\mathbf{S}_{co})_{\sigma_A}$$

$$\Delta\mathbf{OF} = \mathbf{DI}(\mathbf{S}_{ca})_{\sigma_B} - \mathbf{DI}(\mathbf{S'}_{ca})_{\sigma_B} - \mathbf{DI}(\mathbf{S}_{co})_{\sigma_B}$$

Conclusion: If $|\sigma_B|$ **-** $|\sigma_A|$ **= 1, then** Δ**OF can be** < 0 ***(due to*** $-DI(S'_{ca})_{\sigma_B} - DI(S_{co})_{\sigma_B}$***), and the Objective Function can thus deteriorate***).

See Figure 7 for a visual representation of this case.

Only case 1, $|\sigma_B|$ - $|\sigma_A|$ = -1, provides full certainty of how **OF** will evolve. To gain more certainly, we shall also investigate the relationship between the **DI** of service $\mathbf{S}_{ca}$ and the **DI**s of sub services $\mathbf{S'}_{ca}$ and $\mathbf{S}_{co}$.

Relationship between DIs

(1) $\mathbf{DI}(\mathbf{S}_{ca})_\sigma = \frac{\sum_{i=1}^{nca} \mathbf{PF}(\mathbf{p}i)_\sigma}{nca}$ =

$\frac{\sum_{i=1}^{1} \mathbf{PF}(\mathbf{p}i)_\sigma + \sum_{i=j+1}^{j+nco} \mathbf{PF}(\mathbf{p}i)_\sigma + \sum_{i=j+nco+1}^{nca} \mathbf{PF}(\mathbf{p}i)_\sigma}{nca}$

$\implies nca.\mathbf{DI}(\mathbf{S}_{ca})_\sigma - \sum_{i=j+1}^{j+nco} \mathbf{PF}(\mathbf{p}i)_\sigma =$

$\sum_{i=1}^{j} \mathbf{PF}(\mathbf{p}i)_\sigma + \sum_{i=j+nco+1}^{nca} \mathbf{PF}(\mathbf{p}i)_\sigma$

(2) $\mathbf{DI}(\mathbf{S'}_{ca})_\sigma = \frac{\sum_{i=1}^{nca} \mathbf{PF}(\mathbf{p}i)_\sigma + \sum_{i=j+nco+1}^{nca} \mathbf{PF}(\mathbf{p}i)_\sigma}{nca-nco}$

(3) $nco.\mathbf{DI}(\mathbf{S}_{co})_\sigma = \sum_{i=j+1}^{j+nco} \mathbf{PF}(\mathbf{p}i)_\sigma$

Putting (1), (2) and (3) together

$\mathbf{DI}(\mathbf{S'}_{ca})_\sigma = \frac{nca.\mathbf{DI}(\mathbf{S}_{ca}) - \sum_{i=j+1}^{j+nco} \mathbf{PF}(\mathbf{p}i)_\sigma}{nca-nco}$

$\implies \mathbf{DI}(\mathbf{S'}_{ca})_\sigma = \frac{nca}{nca-nco}.\mathbf{DI}(\mathbf{S}_{ca})_\sigma - \frac{nco}{nca-nco}.\mathbf{DI}(\mathbf{S}_{co})_\sigma$

$\implies \frac{nca}{nca-nco}.\mathbf{DI}(\mathbf{S}_{ca})_\sigma = \mathbf{DI}(\mathbf{S'}_{ca})_\sigma + \frac{nco}{nca-nco}.\mathbf{DI}(\mathbf{S}_{co})_\sigma$

$\implies \mathbf{DI}(\mathbf{S}_{ca})_\sigma = \frac{nca-nco}{nca}.\mathbf{DI}(\mathbf{S'}_{ca})_\sigma + \frac{nco}{nca}.\mathbf{DI}(\mathbf{S}_{co})_\sigma$

Let $\alpha = \frac{nco}{nca}$ be the split-factor, where $0 \leq \alpha \leq 1$

Then

$$\mathbf{DI}(\mathbf{S}_{ca})_\sigma = (1-\alpha).\mathbf{DI}(\mathbf{S'}_{ca})_\sigma + \alpha.\mathbf{DI}(\mathbf{S}_{co})_\sigma \qquad (12)$$

This formula expresses $\mathbf{DI}(\mathbf{S}_{ca})_\sigma$ as the linear interpolation between $\mathbf{DI}(\mathbf{S'}_{ca})_\sigma$ and $\mathbf{DI}(\mathbf{S}_{co})_\sigma$, with α as the interpolation factor. See Figure 10 for a visualization of this linear interpolation function.

Two cases are possible:

- Case a: $\mathbf{DI}(\mathbf{S'}_{ca})_\sigma > \mathbf{DI}(\mathbf{S}_{co})_\sigma$
- Case b: $\mathbf{DI}(\mathbf{S}_{co})_\sigma > \mathbf{DI}(\mathbf{S}_{ca})_\sigma$

Based on the relationship between DIs, we may conclude that:

If $|\sigma_B|$ - $|\sigma_A|$ = 1
then Δ**OF=** $\mathbf{DI}(\mathbf{S}_{ca})_{\sigma_B} - \mathbf{DI}(\mathbf{S'}_{ca})_{\sigma_B} - \mathbf{DI}(\mathbf{S}_{co})_{\sigma_B} < 0$
as according to (12) either $\mathbf{DI}(\mathbf{S'}_{ca})_{\sigma_B}$ or $\mathbf{DI}(\mathbf{S}_{co})_{\sigma_B}$ is

$DI(S_{ca})_\sigma = (1-\alpha).DI(S'_{ca})_\sigma + \alpha.DI(S'_{co})_\sigma$

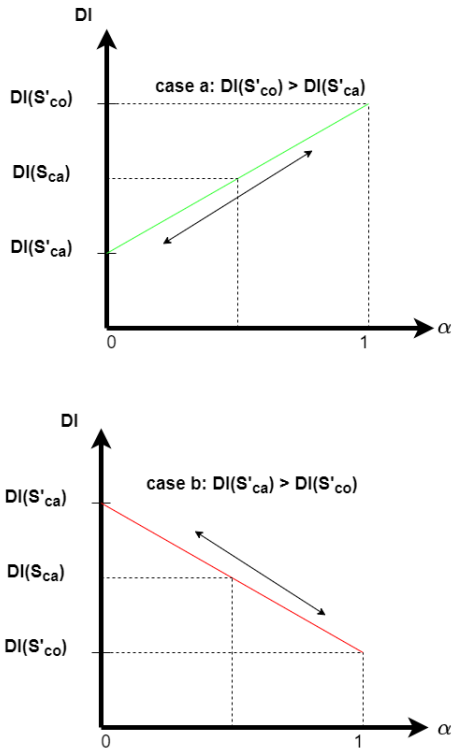


Fig. 10. Linear interpolation

$> \mathbf{DI}(\mathbf{S}_{ca})_{\sigma_B}$.
The Objective Function thus deteriorates when $|\sigma_B|$ - $|\sigma_A|$ = 1.

If $|\sigma_B|$ - $|\sigma_A|$ = 0
then $\Delta\mathbf{OF} = \mathbf{DI}(\mathbf{S}_{ca})_{\sigma_B} - \mathbf{DI}(\mathbf{S'}_{ca})_{\sigma_B} + 1 + \sum_{i=1}^{n} \frac{qvi}{nvi}$ can be < 0
if $\Delta\mathbf{OF} < 0$ then $\mathbf{DI}(\mathbf{S}_{ca})_{\sigma_B} < \mathbf{DI}(\mathbf{S'}_{ca})_{\sigma_B}$ must be ¡ 0
Taking (12) into account, we can rewrite Δ**OF** as:

$(1-\alpha).\mathbf{DI}(\mathbf{S'}_{ca}) + \alpha.\mathbf{DI}(\mathbf{S}_{co}) - \mathbf{DI}(\mathbf{S'}_{ca}) + 1 + \sum_{i=1}^{n} \frac{qvi}{nvi}$

$\implies -\alpha.(\mathbf{DI}(\mathbf{S'}_{ca}) - \mathbf{DI}(\mathbf{S}_{co}) + 1 + \sum_{i=1}^{n} \frac{qvi}{nvi} > 0$

$\implies \Delta\mathbf{OF} > 0$ if $\alpha < \frac{1+\sum_{i=1}^{n} \frac{qvi}{nvi}}{\mathbf{DI}(\mathbf{S'}_{ca}) - \mathbf{DI}(\mathbf{S}_{co})}$

Thus, a smaller α gives a higher probability of an **OF** improvement.

2) Split Selection: From the preceding, we conclude that carving-out subgroups has a high likelihood to result in an improvement of the Objective Function. We will even go a step further and define our move type as the carving-out of all subgroups of a service definition. We call our split operator the full-carve-out move. Example: A service definition $\mathbf{S}$={1,2,3,4,5,6,7}. There also exists service definitions $\mathbf{S_1}$={1,2} and $\mathbf{S_2}$={5,7}. Carving out $\mathbf{S_1}$ and $\mathbf{S_2}$ from $\mathbf{S}$ gives,

$\mathbf{S_1}$= {1,2}, $\mathbf{S_2}$={5,7} and **S'**={3,4,6}.

This move type, however, is unable to handle partial overlapping service definitions. It is expected, therefore, that when all carve-outs are done, there will be a number of overlaps remaining that require a different type of operation.

F. Move Strategy

All services with a **DI** greater than one are candidates for splitting. It seems logical to begin by splitting the service with the largest **DI**. If that service cannot be split (no subgroups), then the second-largest **DI** is taken, etc. If a group can be split, the impact of the split is calculated. When **OF** improves (=descends), the move is accepted and executed. If not, the next service in the sorted service **DI** list is chosen. The move-strategy is a variant of the First Improvement strategy of the ILS meta heuristic; a variant as we first order the service **DI** list and take the top element from the list.

G. Perturbation

The carve-out of subgroups cannot remove all forms for non-disjointness. Correlated (partially overlapping) service definitions cannot be split this way. This creates a requirement for a new split operator when no additional carve-outs are possible. The operator will determine if a service definition overlaps with another service definition. If it does, the intersection is split-off. By splitting off this intersection, a new services definition will be created. Splitting off an intersection will result in $|\sigma_B| - |\sigma_A| = 1$. In the previous section we observed that the Objective Function will deteriorate. This is a transitory situation, due to the fact that the newly-formed service definitions may be subgroups of the existing service definitions. We consciously allow the **OF** temporary deterioration so that a better optimum may be found in the next Local Search iteration. We consider this kind of split as the perturbation.

H. Stop Conditions

Once all possible carve-outs and perturbations are complete, then there are no more inclusively matching and correlated rules. All port frequencies are equal to one, all service group **DI**'s are equal to one, and **OF** will equal the number of service definitions. The solution cannot be additionally improved.

Figure 11 shows how we expect the Objective Function to evolve over time, via consecutive local searches (doing full-carve-out moves) and perturbations (doing intersection-carve-out-moves), until the end condition is reached (i.e., full services are disjoint).

I. Algorithm Overview

The algorithm has been implemented in JAVA. The different components of the solution are implemented as JAVA classes. We attempted to stay as true as possible to NS principles by defining data classes, which only contain data and convenience methods to get and set the data, and task classes used to perform actions and calculations on the data objects. A high level overview of algorithm can be found in Algorithm 1. More

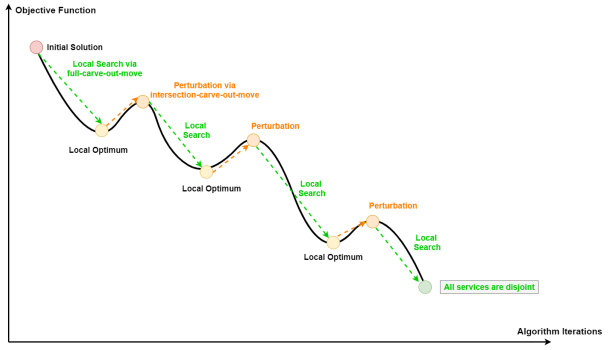


Fig. 11. Expected evolution of the Objective Function

algorithm details and implementation details can be found in [2].

V. Artifact Demonstration

The artifact outlined in the previous section will be applied to operational firewall rule bases provided by Engie. In [1] a manually created test rule base (containing a lot of exceptions to properly test the algorithm encoding) was used to validate the concept and the implementation. We now apply the artifact to operational data to see the real impact it has on a rule base. Before an export from a firewall can be used as input for the algorithm, some pre-processing is required. We start this section by explaining these operations. We continue by discussing the components we added to the algorithm that allow the adjustments to the rule base and thus measure the impact of service disjointness on the size of the rule base. The different demonstration sets will be elucidated before they become subject to the algorithm. We conclude with a summary of the algorithm's results and a description of some in-depth behavioral characteristics of the algorithm.

A. Firewall Export Pre-Processing

Engie provided firewall rule base exports that are implemented on Palo Alto firewalls (a leading manufacturer and provider of firewalls). Those exports required some pre-processing before the brown-field artifact can be used. The pre-processing steps include:

- Remove non-relevant information from the exported CSV files. This is the only manual step.
- Prepare data structures that allow historization (tracking of all changes to the rule base and services during algorithm execution) of the rule base and services.
- Replace firewall group objects that aggregate other group objects and adjust the rules accordingly.
- Adjust the rule base such that each rule only contains one service group.
- Version the rules and services (for historization).
- Remove non-unique services.

B. Adjusting the Rules

Each time a sub-service gets carved-out or an intersection between two services gets split off, adjustments to the rules

base are required. All the rules containing the original service must be adjusted to reflect the result of the split. The rules must also be split on the basis that rules must adhere to our SoC design criteria. Adjustments to the rule base occur at two instances of the algorithm: when a successful sub services carve-out is performed and when a successful overlapping services carve-out is performed.

C. Demonstration Data Sets

Engie provided exports from 15 Palo Alto firewalls in use within Belgium- and Paris-based data centers. The data centers contain multiple firewalls with different filtering strategies. We requested firewall exports that would represent the different types of filtering strategies. Additional contextual information for each firewall can be found below.

- AIMv2: Firewall used to filter in- and outbound traffic of Internet-exposed resources.
- AdminBE: Firewall used to filter traffic between data center client hosting zones and a shared management zone containing services as backup, and monitoring and system management tools. The firewall is located in the Belgium-based data center.
- AdminFR: Idem as AdminBE but for a firewall located in the Paris-based data center.
- AWSDCN: Firewall that acts as a filter between the Engie backbone network and the AWS Direct Connect (dedicated connection to AWS cloud data center in Dublin).
- HOSTING-BE-EBL: Firewall protecting the client hosting zone for Electrabel (a business unit of the Engie Group) in the Belgium-based data center.
- HOSTING-BE-ORES: Firewall protecting the client hosting zone for ORES (a former part of Electrabel, no longer part of the Engie Group), in the Belgium-based data center.
- HOSTING-BE-RAS: Firewall protecting Remote Access Resources, in the Belgium-based data center.
- HOSTING-BE-SHARED: Firewall protecting resources that are shared between various business units of the Engie Group, in the Belgium-based data center.
- HOSTING-BE-TRACTEBEL: Firewall protecting the client hosting zone for TRACTEBEL (a business unit of the Engie Group), in the Belgium-based data center.
- HOSTING-FR-COFELY: Firewall protecting the client hosting zone for COFELY (a business unit of the Engie Group), in the Paris-based data center.
- HOSTING-FR-GRDF: Firewall protecting the client hosting zone for GRFD (a former business unit of the GDF, no longer part of the Engie Group), in the Paris-based data center.
- HOSTING-FR-RAS: Firewall protecting Remote Access Resources, in the Paris-based data center.
- HOSTING-FR-SHARED: Firewall protecting resources that are shared between various business units of the Engie Group, in the Paris-based data center.

Algorithm 1: ILS for service list normalization

```
load_initial_solution (= rule base);
create neighborhood list (= list of services and their DI);
fully_disjoint = FALSE;
end_of_neighborhood = FALSE;
objective_function_improvement = FALSE;
calculate current_objective_function (sum of all service DI of
  neighborhood);
while NOT full_disjoint AND NOT end_of_neighborhood do
    sort neighborhood list (highest DI at top of list);
    neighborhood_pointer = 1 (top of list);
    objective_function_improvement = FALSE;
    while NOT improvement_objective_function AND NOT
      fully_disjoint AND NOT end_of_neighborhood do
        service_to_split = service to which
          neighborhood_pointing is referring;
        perform full-carve-out move on service_to_split;
        calculate new_objective_function;
        objective_function_improvement =
          (new_objective_function <
          current_objective_function?);
        if objective_function_improvement = TRUE then
            reset neighborhood based on full-carve-out
              move;
            reflect full-carve-out move in rule base;
            current_objective_function =
              new_objective_function
            fully_disjoint = (are all service DI of the
              neighborhood = 1);
        else
            neighborhood_pointer ++
        end
        end_of_neighborhood = (neighborhood_pointer
          pointing to last element in neighborhood list?);
    end
    if end_of_neighborhood then
        look for overlapping services in the neighborhood
        if overlapping_services_exists then
            perform intersection-carve-out move;
            reset neighborhood based on
              intersection-carve-out move;
            reflect intersection-carve-out move in rule base;
            calculate new_objective_function;
            current_objective_function =
              new_objective_function
            fully_disjoint = (are all service DI of the
              neighborhood = 1);
            end_of_neighborhood = FALSE;
        else
            end_of_eighborhood = TRUE;
        end
    end
end
if fully_disjoint then
    PRINT "Probably the Global Optimum has been found";
else
    PRINT "Local Optimum found";
end
PRINT RESULT = neighborhood;
```

- IAF: Firewall protecting access between the resources of the user network and data center, via Identify Aware filtering rules.
- IoT-BE: Firewall protecting IoT related resources in the Belgium-based data center.

The demonstration data set also contains an artificially-created rule base entitled Demoset which was used to test the algorithm. Demoset contains as many anomalies as possible to test special conditions that could occur but that are difficult to filter out of the given exports.

D. Demonstration Results

In the following subsections, we review the demonstration environment, the summary table of the demonstrations, and the relationship between the Objective Function and the number of rules in the rule base. We continue with a discussion of the impact of the algorithm on the number of service definitions, and have a closer look at the evolution of the Objective Function during the algorithm's execution. We conclude with an example of how rule and service definition changes are tracked during algorithm execution.

1) Demonstration Environment: The algorithm is written in JAVA using JAVA SDK 1.8.181, developed in the NetBeans IDE V8.2. The demonstration ran on an MS Surface Pro (5th Gen) Model 1796 i5 - Quad Core @ 2.6 GHz with 8 GB of memory, running Windows 10.

2) Demonstration Overview: The algorithm results for the different rule bases can be found in Table I which contains the following information:

- Initial Number of Rules (NoR): number of rules as read from the firewall export files.
- Initial Number of Services (NoS): number of services as read from the firewall export files.
- Initial Number of Service Groups (NoSR): number of service groups as read from the firewall export files.
- Pre-Processing Number of Rules (NoR): number of rules after pre-processing.
- Pre-Processing Number of Unique Services (NoUS): number of unique services after pre-processing.
- Pre-Processing **OF**: the value of the Objective Function, after pre-processing and thus at start of the algorithm.
- Final Number of Rules (NoR): the number of rules after applying the algorithm.
- Final Number of Services (NoS): the number of service definitions after applying the algorithm.
- Final **OF**: the value of the Objective Function after applying the algorithm.

The algorithm performance indicators can be found in Table II.

- Algorithm execution time: time required to disentangle the services and adjust the rules.
- Total execution time: time required to perform the data loading, pre-processing, disentanglement, to print the end result and log, and for the result to be displayed on the screen.
- Level 1 Iterator: number of times the outer loop of the algorithm has run.
- Level 2 Iterator: number of times the inner loop of the algorithm has run.

TABLE I
OVERVIEW DEMONSTRATION RESULTS

Rule Base	Initial			After Pre-Processing			Final		
	NoR	***NoS***	***NoSG***	***NoR***	***NoUS***	**OF**	***NoR***	***NoS***	**OF**
AIMV2	207	250	6	498	226	577.9	1,263	288	288
AdminBe	461	597	41	1,443	547	3,214.1	8,994	547	547
AdminFR	655	717	46	2,584	699	4,469.3	29,377	668	668
AWSDCN	13	13	1	13	13	14.9	22	13	13
Demoset	21	24	8	104	21	44.9	249	34	34
HOSTING-BE-EBL	350	304	10	759	259	877.6	3,841	256	256
HOSTING-BE-ORES	462	336	13	1,306	274	1,205.6	4,936	267	267
HOSTING-BE-RAS	20	16	0	28	16	17.5	29	16	16
HOSTING-BE-SHARED	107	120	7	223	10 7	188.7	360	106	106
HOSTING-BE-TRACTEBEL	10	5	1	10	5	5	10	5	5
HOSTING-FR-COFELY	10	9	1	16	9	9	16	9	9
HOSTING-FR-GRDF	118	46	4	213	42	50	223	40	40
HOSTING-FR-RAS	21	16	1	29	16	17,5	30	16	16
HOSTING-FR-SHARED	198	139	6	359	126	250.2	509	127	127
IAF	32	10	0	34	10	10	34	10	10
IOT-BE	23	28	0	38	25	36.5	47	24	24

TABLE II
PERFORMANCE OF THE ALGORITHM

Rule Base	Execution Information			
	ILS (ms)	Total (ms)	L1	L2
AIMV2	187,227	396,000	25	1,507
AdminBe	520,944	824,000	135	13,609
AdminFR	820,847	1,242,000	154	23,165
AWSDCN	811	18,000	2	3
Demoset	1,358	120,000	22	430
HOSTING-BE-EBL	76,039	265,000	34	2,016
HOSTING-BE-ORES	193,436	632,000	59	2,587
HOSTING-BE-RAS	99	1,000	2	3
HOSTING-BE-SHARED	35,139	202,000	12	427
HOSTING-BE-TRACTEBEL	63	1,000	2	2
HOSTING-FR-COFELY	54	1,000	2	2
HOSTING-FR-GRDF	122	1,000	3	6
HOSTING-FR-RAS	96	1,000	2	36
HOSTING-FR-SHARED	78,503	210,000	19	929
IAF	68	1,000	2	2
IOT-BE	28,731	130.000	3	19

TABLE III
%**OF** IMPROVEMENT VS INITIAL NUMBER OF RULES (NOR)

Rule Base	**Initial NoR**	%**OF** Improvement
HOSTING-BE-TRACTEBEL	10	0%
HOSTING-FR-COFELY	10	0%
AWSDCN	13	10%
HOSTING-BE-RAS	20	9%
Demoset	21	24%
HOSTING-FR-RAS	21	9%
IOT-BE	23	34%
IAF	32	0%
HOSTING-BE-SHARED	107	44%
HOSTING-FR-GRDF	118	20%
HOSTING-BE-SHARED	198	49%
AIMv2	207	61%
HOSTING-BE-EBL	350	71%
AdminBE	461	83%
HOSTING-BE-ORES	462	78%
AdminFR	655	85%

3) Objective Function and the Number of Rules: In Table III and Figure 12, we represent the relationship between the % of **OF** improvement and the number of initial rules in the rule base (NoR).

Three out of the sixteen firewalls contain fully disjoint service definitions: HOSTING-BE-TRACTEBEL, HOSTING-FR-COFELY and IAF. Those are also the firewalls with the fewest rules and service definitions. The algorithm detects the full disjointness and leaves the service definitions and rule base as is.

Six out of the sixteen firewalls are fairly close to having disjoint service definitions: AWSDCN, Demoset, HOSTING-BE-RAS, HOSTING-FR-GRDF and IOT-BE. Those firewalls have a number of rules and services definitions that are below 100 (HOSTING-FR-GRDF having a number of rules a bit above 100). The total improvement of the **OF** is limited to about 25 %.

The remaining eight firewalls contain many more rules and service definitions and the value of the difference between the initial and final value of the **OF** is at least 50 %, with a maximum of 85 %. These numbers confirm that, without proper rule design criteria, the probability of getting a non-evolvable rule base drastically increases with the size of the rule base. The trend between the size of the rule base and the percentage of **OF** improvement (a good indicator for the status of the initial evolvability), should be an asymptotic function trending toward 100 %. A logarithmic regression provides a good fit.

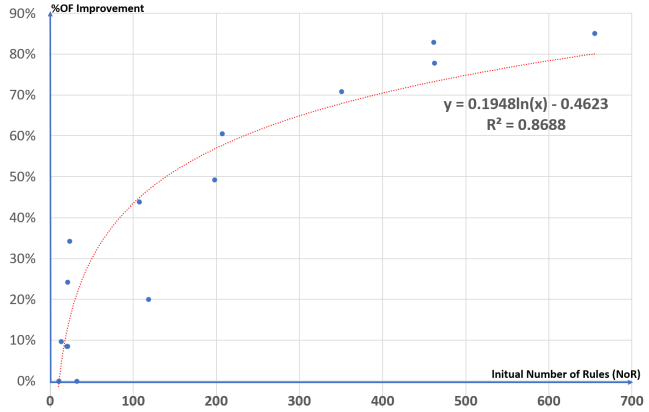


Fig. 12. %**OF** Improvement vs number of rules in the rule base

In Table IV and Figure 13, we represent the relationship between the % **OF** improvement and the % of extra rules (growth rule base) due to the service disentanglement algorithm.

4) Impact of the Algorithm on the Number of Service Definitions: Upon examination of the number of service definitions at the end of the algorithm, we note that splitting the service definitions does not have a large impact on the total number of services. See Figure 14 for an overview. There is even a tendency toward the total number of service definitions decreasing slightly. It seems to be that the algorithm rearranges the ports into more suitable groups, without having the number of service definitions proliferate.

5) Evolution of the Objective Function During Algorithm Execution: To visualize what occurs during the algorithm execution, three indicators are tracked: the **OF**, the outer loop iterations, and the inner loop iterations. The "Level 1 Indicator" is the number of times that the outer **DO** loop of the algorithm has run. The indicator measures the number of times a perturbation or successful carve-out is done. The "Level 2 Indicator" (L1I) is the number of times the inner **DO** loop of the algorithm runs within a given number of level 1 iterations. Each time the "Level 1 Iterator" increments, the "Level 2 Iterator" (L2I) is reset. We plot the evolution of these three indicators against the cumulative number of level 2 iterations for two of the firewalls with a number of rules below 100, in Figure 15 and Figure 16. In Figure 17 and Figure 18 we show the evolution of the three indicators for two firewalls containing in excess of 100 rules.

TABLE IV
% **OF** IMPROVEMENT VS %GROWTH RULE BASE

Rule Base	%OF Improvement	%Growth Rule Base
HOSTING-BE-RAS	9%	4%
HOSTING-FR-RAS	9%	3%
AWSDCN	10%	69%
HOSTING-FR-GRDF	20%	5%
Demoset	24%	139%
IOT-BE	34%	24%
HOSTING-BE-SHARED	44%	61%
HOSTING-FR-SHARED	49%	42%
AIMv2	61%	154%
HOSTING-BE-EBL	71%	406%
HOSTING-BE-ORES	78%	278%
AdminBE	83%	523%
AdminFR	85%	1037%

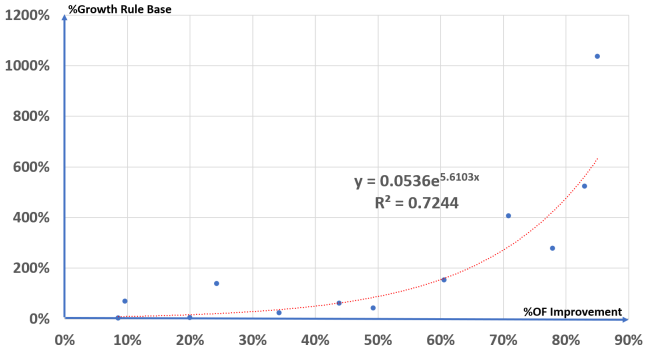


Fig. 13. % extra rules vs %Δ**OF**

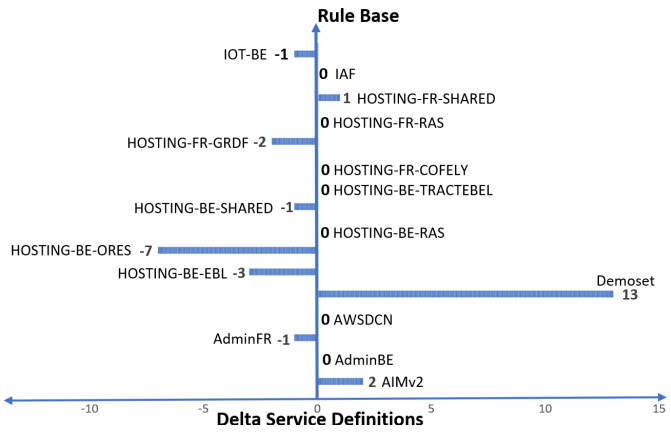


Fig. 14. Impact of the algorithm on number of services.

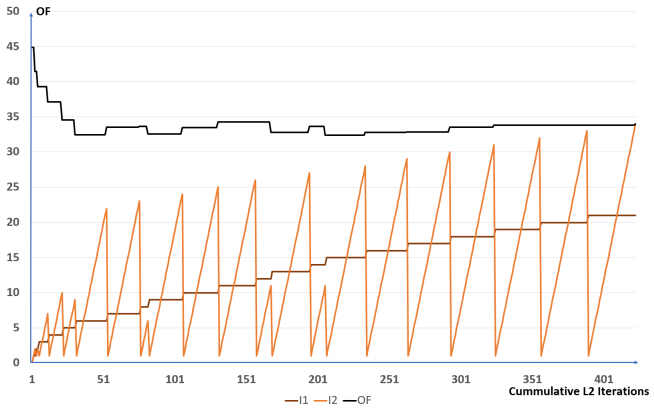


Fig. 15. **OF**, L1 and L2 for the Demoset firewall.

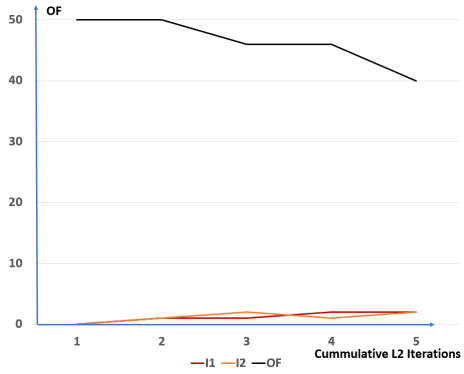


Fig. 16. **OF**, L1 and L2 for the HOSTING-FR-GRDF firewall.

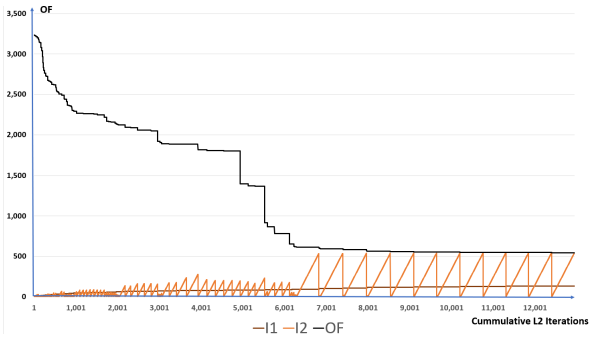


Fig. 17. **OF**, L1 and L2 for the AdminBE firewall.

6) Tracking of Rule and Service Definition Changes: The algorithm tracks all changes that are made to the rules. As an example, the log excerpt below shows the evolution of rule 6 from the Demoset, as provided by the algorithm at end of execution.

- During pre-processing, the Service Group ”SERVICE25”, is replaced by its members ”SERVICE17” and ”SERVICE19”. The rule now has 6.1 as identifier.
- During pre-processing, the rule is split into two rules, 6.1.1 and 6.1.2 since rule 6.1 was contained in two service definitions.
- Duinrg pre-processing, rules 6.1.1 and 6.1.1.2 get the versioned service definitions. At this point, rule 6 is replaced by 6.1.1.1 and 6.1.2.1.
- During the ILS, the service ”SERVICE17 V0” gets split into ”Service 17 V0.2” and the existing service ”SERVICE19 V0”, and the rule 6.1.1.1 splits into 6.1.1.1.1 and 6.1.1.1.2.

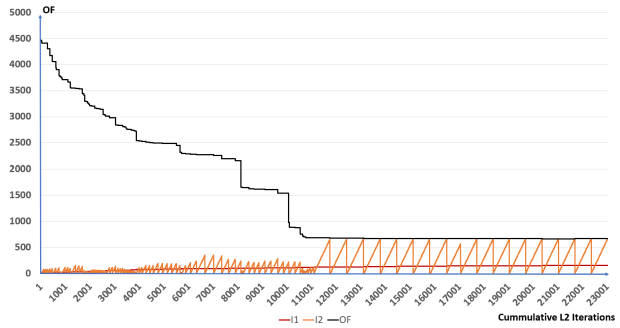


Fig. 18. **OF**, L1 and L2 for AdminFR firewall.

```
6;R6;SERVICE25,
*6.1;R6.1;SERVICE17,SERVICE19,
**6.1.1;R6.1.1;SERVICE17,
***6.1.1.1;R6.1.1.1;SERVICE17 V0,
****6.1.1.1.1;R6.1.1.1.1;SERVICE19 V0,
****6.1.1.1.2;R6.1.1.1.2;SERVICE17 V0.2,
**6.1.2;R6.1.2;SERVICE19,
***6.1.2.1;R6.1.2.1;SERVICE19 V0,
```

In summary, rule 6 was replaced by rules 6.1.1.1.1, 6.1.1.1.2 and 6.1.2.1.

The evolution of the services is tracked in a similar manner. In the log excerpt below, the evolution of ”SERVICE17” and ”SERVICES19” is shown (versioning, splitting).

```
SERVICE17;UDP;40-41
*SERVICE17 V0;UDP;40-41
**SERVICE19 V0;UDP;40
**SERVICE17 V0.2;UDP;41

SERVICE19;UDP;40
*SERVICE19 V0;UDP;40
```

VI. Evaluation and Discussion

This section starts with evaluating the functioning of the artifact followed by the analysis of the worst case number of operations required to complete the algorithm and the comparison with the artifact performance. We continue with pondering on the question whether the artifact has found the most optimal solution, and by analysing the impact of the algorithm on size of the rule base and firewall scaling. We continue with discussing the entropy present in a rule base and how the algorithm measures this entropy. We finish by positioning the artifact in a firewall management tool.

A. Algorithm functioning

Based on the demonstration, we can conclude that an algorithm based on an ILS meta-heuristic disentangles service definitions and is able to adjust the rule base accordingly. The algorithm is an essential building block in a solution that can

convert an existing firewall rule base into a rule base that is fully compliant with the green-field artifact.

It is an essential building block but not the sufficient building black. It is possible that fully-overlapping rules emerge during the algorithm execution.

Example:

Take a rule R1 that has C1 as source, H1 as destination, and S1 as service.

Now take a rule R2 that has C1 as source H1 as destination, and S2 as source.

Let's consider that S1 and S2 overlap. The overlapping service is S3

Applying the algorithm would give:

– R1: C1 H1 S'1
– R2: C1 H1 S3
– R3: C1 H1 S'2
– R4: C1 H1 S3

As can be seen, R2 and R3 become identical rules which still need to be filtered out.

The demonstration has provided insight into how the Objective Function evolves during algorithm execution, as well as into the relationships between the number of initial rules in the rule base and the corresponding value of the objective function, and the number of rules at the end of the algorithm execution and the change in Objective Function.

B. Big O of the Artifact

The **Big O** of an algorithm expresses the algorithm's complexity, calculated based on the worst-case scenario in terms of the number of operations required in function of the size of the problem to be solved. This formula reflects the worst-case effort required to complete the algorithm execution. The algorithm contains two nested loops that both can iterate over the full neighborhood, meaning the algorithm will be quadratic with respect to the size of the neighborhood. The number of operations performed in the innermost loop, such as Service_DI_list_Creator, Service_split_Evaluator are also proportional to the size of the neighborhood.

We may thus conclude that the **Big O** of the complete algorithm is cubic - $O = n^3$, where **n** is the size of the neighborhood (= size of the solution = the number of service definitions).

C. Performance of the Artifact

Algorithm execution time is measured as the time it takes to disentangle the services after pre-processing. Figure 19 shows the relationship between the initial size of the neighborhood (number of unique services) and algorithm execution time. The exponent of the power function is a bit above two. This is consistent with the **Big O**, where we expected a worst-case exponent of three.

Measures could be taken to ensure better algorithmic performance. The innermost loop iterates over all services until it locates one that contains subgroups. All services that already have a **DI** of 1 should not be further investigated. As the neighborhood is sorted from high to low **DI** at the start of the inner loop, the inner loop could stop as of the first service where a **DI** of 1 is encountered. According to meta-heuristics, this value represents a form of algorithmic memory, indicating parts of the neighborhood that can no longer improve and should thus not be investigated.

TABLE V
ARTIFACT PERFORMANCE

Rule Base	Number of Unique Services (NoUS)	Algorithm Execution Time (ms)
HOSTING-BE-TRACTEBEL	5	63
HOSTING-FR-COFELY	9	54
IAF	10	68
AWSDCN	13	811
HOSTING-BE-RAS	16	99
HOSTING-FR-RAS	16	96
Demoset	21	1,358
IOT-BE	25	28,731
HOSTING-FR-GRDF	42	122
HOSTING-BE-SHARED	107	35,139
HOSTING-FR-SHARED	126	78,503
AIMv2	226	187,227
HOSTING-BE-EBL	259	76,039
HOSTING-BE-ORES	274	193,436
AdminBE	547	520,944
AdminFR	699	820,847

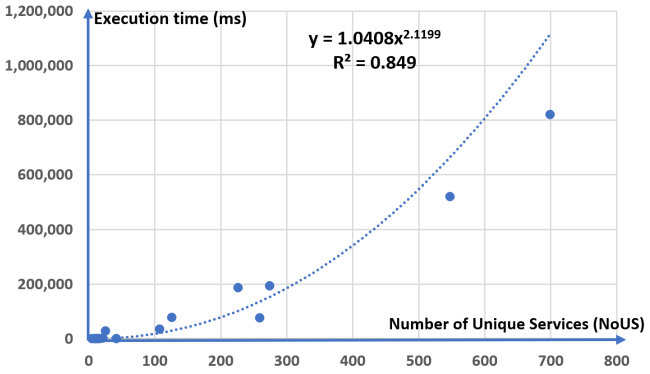


Fig. 19. Artifact performance

D. Global Optimum

Does the heuristics-based algorithm establish the Global Optimum? It is quite difficult to formally prove that heuristic algorithms always provide the most optimal solution. After all, the full solution space of all possible groups combining all possible ports is exponential (see combinatorics) and quickly becomes impossible to fully search.

We do think that, given the initial solution, we have succeeded in converging on the most optimal solution. Sub-optimal solutions always will have either subgroup and/or overlapping groups. The algorithm filters out all subgroups in the inner loop and, if no additional subgroups are found, it searches for overlaps, after which it again scans for subgroups. As both inner and outer loop iterations search the entire neighborhood, all possible subgroups and overlaps are located and eliminated. While we are not presently able to provide formal proof, we nonetheless believe that, from a given initial solution, the set of services that are disjoint and maximum in size is found.

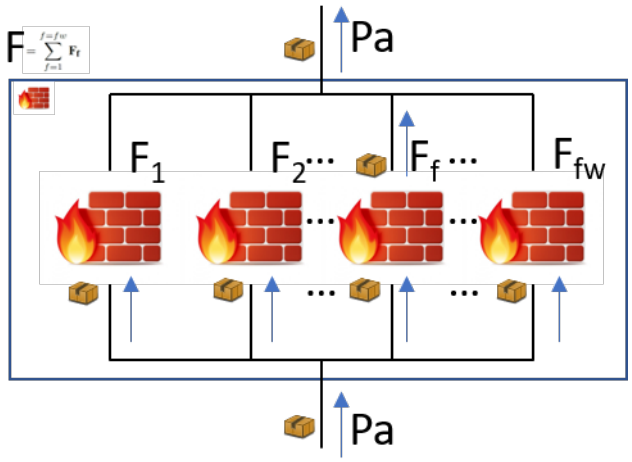


Fig. 20. Scaling of firewalls with an evolvable rule base

E. Implication of the Artifact on Firewall Rule Base Size and Scaling

In all application of NS on design of systems, the result is a more fine grained structure [7] [17] [18]. Applying NS on a firewall, and thus making it more evolvable, is no exception. From the previous section we can clearly see that the amount of extra rules can increase considerably. This brings forward the question of firewall performance - how quickly will the firewall find the rule to apply to the traffic.

In an evolvable rule base, all the rules are disjoint from one another and each network package can only hit one rule. This rule can be located in the beginning or near the end of the rule base. As there is only one rule that can be hit, the rule base may be split into multiple parts and distributed in parallel across different firewalls.

Let **F** be a firewall rule base containing only disjoint rules created according to the green-field artifact. As visualized in Figure 20, **F** can be split into **f**w sub rule bases, which are spread over **f**w parallel firewalls. Each of the **f**w rule bases conclude with the “Default Deny” rule.

A network package will attempt to traverse each firewall, but only one of the firewalls has a rule it can hit.

$$\mathbf{F} = \sum_{f=1}^{f=fw} \mathbf{F}_{\mathrm{f}}$$

Let $\phi_{\mathbf{f}}(\mathbf{F_f},\mathbf{Pa})$ be the firewall filtering function that takes rule base $\mathbf{F_f}$ and package **Pa** as input.

- $\phi_{\mathbf{f}}(\mathbf{F_f},\mathbf{Pa})$ = 0 if the package is blocked - there is no rule **R** in $\mathbf{F_f}$ such that the package is allowed
- $\phi_{\mathbf{f}}(\mathbf{F_f},\mathbf{Pa})$ = 1 if the package is allowed - there is a rule **R** in $\mathbf{F_f}$ such that the package is allowed

Let $\Phi^{\mathbf{P}}{}_{\mathbf{fw}}$ be the parallel firewall filtering function. Then:

$$\Phi^{\mathbf{P}}{}_{\mathbf{fw}}(\mathbf{PA}) = \sum_{f=1}^{f=fw} \phi_{\mathbf{f}}(\mathbf{F_f},\mathbf{Pa})$$

Where:

- $\Phi^{p}_{fw}(\mathbf{Pa})$ = 0, if **Pa** is blocked by all of the fw firewalls.
- $\Phi^{p}_{fw}(\mathbf{Pa})$ = 1 if **Pa** is allowed by one of the fw firewalls.
- $\exists! F_f \in F$ for $f = 1 \rightarrow fw \implies | \, R \in F_j$

A rule base that exclusively contains disjoint rules can scale horizontally (i.e., employ parallel firewalls). Firewalls with a non-evolvable rule base can only scale vertically (i.e., employ a larger firewall). Scaling, however, does not come without significant cost. Modern firewalls allow virtualization, but each virtual instance comes at a cost as well.

In addition to the horizontal scaling possibilities of an evolvable rule base, the performance of an evolvable rule base can be boosted by moving the most frequently used rules to the top. Check Point, a firewall vendor, suggests locating the rules that are most frequently hit (and applied) at the top of the firewall table. In a rule base that is order-sensitive, this is a real issue. In a rule base that is not order-sensitive, one could monitor the firewall to determine which rules are hit most and then prioritize those rules without having to worry about the potential impact to other rules. Doing this dynamically would be even more powerful as the firewall would be able to reorganize its rules according to variable daily traffic.

F. Measuring the entropy of a filewall rulebase

We will now examine the question ”What is the impact of the service group disjointness level of a rule base on the size of the aforementioned rule base after application of the brown-field artifact?”. We shall define the Services Disjointness Index (**SDI**) as the ratio between the value of the objective function **OF** and the number of services **S**.

$$\mathbf{SDI} = \frac{\mathbf{OF}}{\mathbf{S}}$$

SDI is 1 in a rule base exclusively containing disjoint services and greater then 1 if the rule base contains non-disjoint services. We would like to know whether or not we may determine the increase in number of rules as a result of the application of the brown-field artifact, based on the initial value of **SDI**.

The **SDI** is a fairly accurate measure for the statistical entropy of a rule base. The macro-state is the number of services in a rule base, the micro-states being the number of possible services within a rule base. An evolvable and perfectly stable rule base would have a ratio of micro-states to macro-state equalling 1. There are multiple configurations of services that deliver a statistical entropy of 1. We are aware of at least two: one port per service, and the one we discovered with the brown-field algorithm by disentangling the services. The **SDI** is, however, an imperfect representation of the statistical entropy of the rule base. Indeed, we have demonstrated that the brown-field algorithm may result in shadowing rules. An additional operationalization to measure this would be required in order to fully express the statistical entropy of a rule base.

TABLE VI
RNIR VS **SDI**

Rule Base	SDI	RNIR
HOSTING-BE-TRACTEBEL	1	0
HOSTING-FR-COFELy	1	0
IAF	1	0
HOSTING-BE-RAS	1.0938	0.0357
HOSTING-FR-RAS	1.0938	0.035
AWSDCN	1.1069	0.6923
HOSTING-FR-GRDF	1.1905	0.0469
IOT-BE	1.4600	0.2368
HOSTING-BE-SHARED	1.7632	0.6143
HOSTING-FR-SHARED	1.9853	0.4178
Demoset	2.1377	1.3942
AIMv2	2.5573	1.5361
HOSTING-BE-EBL	3.3882	4.0606
HOSTING-BE-ORES	4.3999	2.7795
AdminBE	5.8759	5.2328
AdminFR	6.3938	10.3688

In our experiment, the independent variable is **SDI** and the dependent variable is the relative increase in the number of rules due to application of the treatment (i.e., application of the brown-field artifact). The relative increase in the number of rules (**RINR**) is calculated as the difference between the number of rules (**NR**) after and before application of the artifact, divided by the number of rules before application of the artifact.

$$\textbf{RINR} = \frac{NR_{after} - NR_{before}}{NR_{before}}$$

The result can be found in Table VI and Figure 21. The correlation between the independent and dependent variable is 0.9257.

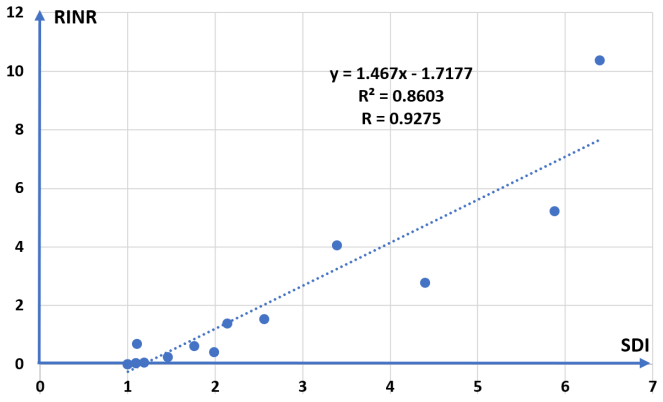


Fig. 21. **RINR** vs **SDI**

G. The Firewall Rule Base Analyser and Normalizer System

As the firewall provides considerable design freedom which could potentially lead to evolvability issues, firewall management should be undertaken outside of the firewall, ideally using a specialized tool that incorporates the artifacts discussed in this dissertation. We characterize this tool as a Firewall Rule Analyser and Normalizer System or FRANS (see Figure 22). Such a tool would ideally have the following features:

- Enforces the usage of the green-field artifact.
- Analyzes an existing rule base — measures disjointedness levels — with the brown-field artifact.
- Converts an existing rule base into an evolvable rule base using the brown-field artifact.
- Will centrally manage all definitions: services, sources, destinations, rules.
- Provides full traceability on all changes performed on the definitions.
- Makes firewalls scale horizontally.
- Changes the rule order dynamically to increase performance.

All firewall rule management activities are done in the tool as opposed to via firewall management consoles. As modern firewalls publish their management functionalities via APIs, the tool can use these APIs to change rules and objects.

The creation of the fine-grained rule base by humans is an issue. The green-field artifact defines criteria for groups and rules that need to be followed strictly. The creation of a catalog of all possible services is required. For standard services and tools, lists of assigned ports/protocols and international standardization organizations related to the Internet (e.g., iana.org) exist and may be reused.

FRANS should expand the firewall rules in the fine-grained format, in accordance with the naming conventions. Checks must also be performed against the group definitions and content in accordance with the green-field artifact and via a user-friendly interface. With this configuration, the tool could then push the rules towards the firewall, which would effectively separate the management from the implementation of rules. Such tools exist on the market. Examples include Algosec, Tufin, Firemon. However, none of those tools consciously restrict the design space for the purpose of enforcing the creation of an evolvable rule base.

While defining a rule for each service may be considered cumbersome, it is possible to create roles such as ”monitoring and management” (i.e., establishing which is a grouping of smaller, disjoint services) in order to mitigate this. In this example, the firewall administrator could create a rule specifying this ”monitoring and management” role to express that the server needs to allow access to all monitoring and management services. The tool would ideally expand these roles into the individual rules for each disjoint service. Examples:

- ”Monitoring and Management” = SSH + SFTP + FTP + SMTP + TELNET
- Host = x
- Rule : C_Hx_SMaM; Hx_S_MaM; S_MaM; allow
- Will be expanded to :
  - C_Hx_S_SSH; Hx_S_SSH; S_SSH; allow
  - C_Hx_S_SFTP; Hx_S_SFTP; S_SFTP; allow
  - C_Hx_S_FTP; Hx_S_FTP; S_FTP; allow

– C_Hx_S_SMTP; Hx_S_SMTP; S_SMTP; allow
– C_Hx_S_TELNET; Hx_S_TELNET; S_TELNET; allow

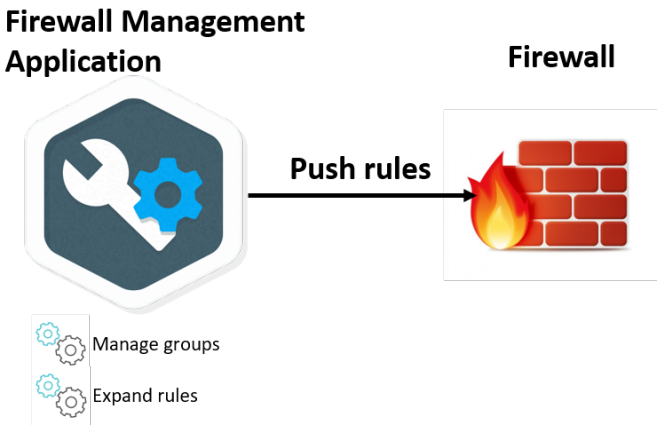


Fig. 22. Firewall management tool

The brown-field artifact should be included in the tool to read and analyze an existing rule base. The Disjoint Index of all groups and the total value of the Objective Function can be calculated. These are important indicators for the level of evolvability and the impact the firewall normalization process will have on the size of the rules base. Highly non-evolvable rule bases may require additional firewall infrastructure to allow horizontal scaling. The tool could create new firewall instances on a virtual infrastructure or spin up new cloud based firewalls on a cloud platform.

FRANS should convert existing rule bases into evolvable rule bases and deploy those on the firewall infrastructure. As FRANS should be a central firewall management platform, it could compare at all times the defined policy in the tool to the active policy on the firewall. This would allow detection of rule adjustments made directly on the firewall and even make firewall rule bases immutable. In FRANS, additional information reflecting why rules are deployed and links with application management tools could be made in order to allow centralized and easily understandable security documentation.

VII. Conclusion

In this section, we will summarize important conclusions. We start with pointing out that the artifact allows the measuring of the evolvability of the firewall. We continue by stating that the artifact leads to a fine-grained rule base and by stating that such fine-grained rule bases allow true horizontal scaling. We finish be pointing out some limitations, lack of comparison with related literature and proposing future work.

A. Measuring the Evolvability of a Firewall

We were able to operationalize one aspect of the evolvability of a firewall, namely the need for disjoint services. Independent from the meaning and functions of the various service ports within the rule base, the **SDI** is an important indicator for the evolvability of the firewall. If **SDI** is greater than 1, the door is left open to the creation of non-evolvable rules. The **SDI** represents the statistical entropy of the service configurations in a rule base and is a good proxy for the statistical entropy of the rule base.

To measure all aspects of evolvability and statistical entropy in accordance with the green-field artifact, a second index concerning the destinations would need to be developed.

B. Impact of the Brown-Field Artifact on the Size of the Rule Base

As SoC has been meticulously applied, the choice of a fine-grained rule base is unsurprising. The relationship between the level of service disjointness and extra rules has been investigated. Additional runs of the algorithm with firewalls from different companies would provide further insight into the complexity of this relationship (and establish whether it is linear, polynomial or exponential).

C. Firewall Scaling

The artifacts produce a fine-grained rule base. A large number of rules in a rule base will have a detrimental impact on performance. But creating an evolvable rule base also provides the answer to this problem, given that only an evolvable rule base will scale truly horizontally.

D. Artifact Limitations

The artifact tracks all changes in the service definitions by means of continuously changing the name of the services via a versioning mechanism. Although the end result is disjoint services according to the green-field artifact, the naming of those services is not compliant with the naming convention put forward in the green-field artifact. A mechanism to generate meaningful names is currently lacking. According to the green-field artifact, we should add a rules in the rule base for each host-service combination. The current version of the brown-field artifact only disentangles the services. Although this leads to disjoint rules, it can still lead to CE [2]. It is quite straight forward to add this step as it just a matter of splitting rules to make sure they only contain one destination and not an aggregation of destinations. The brown-field artifact can result in identical rules and so can the above mentioned splits in destinations. The algorithm does not filter those out. An extra pass through the rule base is required to eliminate those.

E. Related Literature

A more detailed literature study related to this work can be found in [2]. We would however like to stress that, to the best of our knowledge, no other work has been found that addresses the evolvability issues of the firewall. Over the past 40 years, sufficient research has been done boosting performance of the firewall and on problems with firewall management. We have not found work that does an analysis based on a grounded theory of evolvability such as NS. The concept of true horizontal scaling of firewall has also not been observed in literature. The closest we found is work that put an amount of firewalls in parallel but all firewalls contain the

same rule base [23]. This will indeed boost performance as each firewall has less traffic to handle but each firewall still has the same rule base size. In our solution, the size of the rule base can be reduced by spreading it over multiple firewalls.

F. Future Work

This work is an important yet incomplete step toward the evolvable TCP/IP firewall. The green-field artifact needs to be converted into software that will take a high-level security requirement as input, "expand" it into the required fine-grained rules, and push the rules to a firewall. The artifact discussed in the paper (the brown-field artifact) needs to be extended to re-organize the destinations and sources in accordance with the green-field artifact, and requires a solution to naming services such that they are in line with the green-field artifact.

While the requisite groundwork has been established, the remainder needs to be built. We thus regard this not as future research, but rather as future work.

This work has limited itself to the TCP/IP based firewall, which provides a basic security layer like decent locks on the doors and windows of a house. Other security devices, such as application level firewalls, operate at other layers of the OSI stack, above TCP/IP. Filtering rules are installed there as well and if again the filtering rules overlap or contradict, evolvability issues may appear. It is also native to only rely on application level firewalls and no longer on TCP/IP based firewalls, as it is like removing locks from doors and windows and only react on camera surveillance - by the some something shows up on camera, the damage can already be done. Multi level security applies in both the physical and virtual world.

References

[1] G. Haerens, "Usage of iterated local search to improve firewall evolvability", PATTERNS-2021, 2021.

[2] G. Haerens, "On the Evolvability of the TCP-IP Based Network Firewall Rule Base", 2021, PhD Thesis, ISBN: 978-90-5728-716-9, University of Antwerp, 2021

[3] H. Shel and A. Spiliotes, "The State of Network Security: 2017 to 2018", Forrester Research, November 2017

[4] "2018 State of the firewall", Firemon whitepaper, URL https://www.firemon.com/resources/, [retrieved: April, 2021]

[5] "Firewall Management - 5 challenges every company must address", Algosec whitepaper, URL https://www.algosec.com/resources/, [retrieved: April, 2021]

[6] G. Haerens and H. Mannaert, "Investigating the Creation of an Evolvable Firewall Rule Base and Guidance for Network Firewall Architecture, using the Normalized Systems Theory", International Journal on Advances in Security, Volume 13 nr. 1&2, pp. 1-16, 2020

[7] H. Mannaert, J. Verelst and P. De Bruyn, "Normalized Systems Theory: From Foundations for Evolvable Software Toward a General Theory for Evolvable Design", ISBN 978-90-77160-09-1, 2016

[8] E. Al-Shaer and H. Hamed, "Taxonomy of conflicts in network security policies", IEEE Communications Magazine, 44(3), pp. 134-141, March 2006

[9] E. Al-Shaer, H. Hamed, R. Boutaba and M. Hasan, "Conflict classification and analysis of distributed firewall policies", IEEE Journal on Selected Areas in Communications (JSAC), 23(10), pp. 2069-2084, October 2005

[10] M. Abedin, S.Nessa, L. Khan and B. Thuraisingham, "Detection and Resolution of Anomalies in Firewall Policy Rules", Proceedings of the IFIP Annual Conference Data and Applications Security and Privacy, pp. 15–29, 2006

[11] E. Al-Shaer and H. Hamed, "Design and Implementation of firewall policy advisor tools", Technical Report CTI - techrep0801, School of Computer Science Telecommunications and Information Systems, DePaul University, August 2002

[12] S. Hinrichs, "Policy-based management: Bridging the gap", Proceedings of the 15th Annual Computer Security Applications Conference, pp. 209-218, December 1999

[13] A. R. Hevner, S. T. March, J. Park and S. Ram, "Design Science in Information Systems Research", MIS Quarterly, Volume 38, Issue 1, pp. 75-105, 2004

[14] P. Johannesson and E. Perjons, "An Introduction to Design Science", ISBN 9783319106311, 2014

[15] W. R. Stevens, "TCP/IP Illustrated - Volume 1 - the Protocols", Addison-Wesley Publishing Company, ISBN 0-201-63346-9, 1994

[16] H. Zimmermann and J. D. Day, "The OSI reference model", Proceedings of the IEEE, Volume 71, Issue 12, pp. 1334-1340, Dec 1983

[17] H. Mannaert, J. Verelst and K. Ven, "The transformation of requirements into software primitives: Studying evolvability based on systems theoretic stability", Science of Computer Programming, Volume 76, Issue 12, pp. 1210-1222, 2011

[18] P. Huysmans, G. Oorts, P. De Bruyn, H. Mannaert and J. Verelst, "Positioning the normalized systems theory in a design theory framework", Lecture notes in business information processing, ISSN 1865-1348-142, pp. 43-63, 2013

[19] G. Haerens, "Investigating the Applicability of the Normalized Systems Theory on IT Infrastructure Systems, Enterprise and Organizational Modeling and Simulation", 14th International workshop (EOMAS) 2018, pp. 23-137, June 2018

[20] M. Rafael, P. M. Pardalos and M. G. C. Resende, "Handbook of Heuristics", ISBN 978-3-319-07123-7 IS, 2018

[21] Z. Michalewicz and D. B. Fogel, "How to Solve It: Modern Heuristics", ISBN 978-3-642-06134-9, 2004

[22] E. Talbi, "Metaheuristics - From Design to Implementation", ISBN 978-0-470-27858-1, 2009

[23] K. Salah, P. Calyam, R. Boutaba, "Analytical model for elastic scaling of cloud-based firewalls", IEEE Transactions on Network and Service Management, 2016, 14.1: 136-146.

Simulation and Analysis of Tracing Damage Paths and Repairing Objects in Critical Infrastructure Systems

Justin Burns, Brajendra Panda, and Thanh Bui
Computer Science and Computer Engineering Department
University of Arkansas
Fayetteville, AR 72701 USA
email: {jdb083, bpanda, tbui}@uark.edu

***Abstract*—Critical infrastructure systems have recently become more vulnerable to attacks on their data systems through internet connectivity. If an attacker is successful in breaching a system's defenses, it is imperative that operations are restored to the system as quickly as possible. This research focuses on damage assessment and recovery following an attack. We review work done in both database protection and critical infrastructure protection and establish our own definitions of how damage affects the relationships between data and software. Then, we propose a model using a graph construction to show the cascading effects within a system after an attack. We also present an algorithm that uses our graph to compute an optimal recovery plan that prioritizes the most important damaged components first so that the vital modules of the system become functional as soon as possible. This allows for the most critical operations of a system to resume while recovery for less important components is still being performed. Lastly, we show results from simulations using our algorithm on data graphs with various parameters.**

Keywords-critical infrastructure; damage assessment; recovery.

I. INTRODUCTION

Critical infrastructure systems are those that are considered extremely critical to the functioning of a government or a country. As described in [2], critical infrastructures are like the vital organs of a body that need to perform their own roles for the human body to function efficiently and painlessly. The US Department of Homeland Security [3] declares that such systems are "so vital to the United States that their incapacity or destruction would have a debilitating impact on our physical or economic security or public health or safety." Therefore, the protection and smooth functioning of our nation's critical infrastructures are indispensable and cannot be ignored.

These systems are becoming prime targets of attackers – primarily state actors and organized crime – and a major attack on one can cripple the economy of the victim nation. These systems are also more likely to be connected to the internet now to provide benefits like cost reduction (where large systems can be remotely managed over the public network), increased capability (by providing sufficient computing resources for infrastructure hardware with less capability power), and improved efficiency and transaction speed. This connectivity unfortunately makes it easier for attackers to hack into these systems. Consider the New York Times report about the attack on Colonial Pipeline [4]. While the details of the attack are not yet disclosed, a group of cybercriminals was able to compromise data systems using the internet, which resulted in Colonial Pipeline shutting down their pipeline. This outage affected mass transit and other industries across the entire U.S. East Coast and exposed a lack of preparation for such a crisis. This illustrates how an external system can have a relationship with a critical infrastructure system and how such relationships can be exploited to carry out an attack.

It is clear from past incidents and recent reports [5]-[8], (to cite just a few) that attacks on critical infrastructures are occurring frequently, which indicates that prevention mechanisms are not enough to stop them. Thus, it is of utmost importance to aggressively prepare for post-attack activities, which include damage assessment and recovery mechanisms that are critical to making the affected systems available at full functioning mode as soon as possible. This research aims at meeting this important goal.

We propose a framework that models damage spread within a set of data objects based on object dependencies and prioritizes making repairs to the most critical objects first. The framework is based on some of the models explored in critical infrastructure protection and uses a version of previously proposed repair methods that is modified to focus on meeting specific goals when determining the order in which repairs are made.

The rest of the paper is an extension of the work in [1] and is organized as follows. Section 2 offers some work performed in this area. Section 3 defines the problem that we aim to build our model for. In section 4, the types of relationships that exist in data systems are explained. We provide details on our model in section 5, which includes three subsections to explain our definitions, model description, and algorithm. We then show experiments using our model and present the results in section 6. Section 7 concludes our work.

II. RELATED WORKS

This paper aims to examine methods and frameworks used for database and critical infrastructure protection and apply them towards protecting a set of data objects. This section describes some of the publications that are relevant to our proposed framework. Various types of critical infrastructure objects are described in [17]. The focus of our work falls under network and network nodes, which are defined as a "structure with one dominating dimension", and

junctions within the network, respectively. Furthermore, [18] establishes the concept of resilience, a cyclical process by which a system undergoes recovery, adaptation, and prevention between attacks. Turning this concept into a concrete value that can be used in risk assessment has been a point of interest in protecting systems [20]. Efforts have also been made to quantify vulnerability as a measurement that can be used to determine how frequently or severely a system can be at risk [19]. One of the major works on damage assessment and recovery within a database uses data dependency to find data affected by an attack to optimize recovery [9]. While this method relies on the direct relationships between data items, an alternate model to recover data from an attack instead uses the transaction log for assessment [10].

Kotzanikolaou et al. describe a model in [11] that assists in risk assessment for possible scenarios that can result in cascading failures within a CI system. For critical infrastructures with data-rich operations, the use of Cyber-Physical Systems can cause new vulnerabilities as described in [12]. Their model analyzes threats that can appear due to these vulnerabilities and analyzes the potential cascading damage they can cause. System dynamics modeling can also be used to analyze disruptive events to characterize such disruptions to critical infrastructure by risk assessment and various impact factors as shown in [13].

Rehak et al. [14] model an infrastructure system as elements and linkages with different types of relationships establishing dependencies and interdependencies. They note that these elements can have varying criticality, causing some elements to cascade more damage into the system than others in the event of a failure. This work is important because by establishing criticality, they quantify damage within a system. We use this concept of criticality later in this paper to direct the optimal repair path of data objects.

We also consider models that assist with recovery during an attack. In [15], an algorithm is proposed to restore damaged element paths by recursively breaking down demand flows into simpler problems. They use a centrality metric to rank damaged nodes and determine which ones should be repaired first and expand on the use of centrality to make repair decisions in further work [16]. We use the concept of centrality to rank data objects in a case where two or more are equally critical. In our algorithm, we also utilize their method of simplifying damage paths to find the fastest route to restoring intermediate data objects. However, the novelty of our approach is twofold: we must repair all components within the system because data objects cannot have computations rerouted, unlike the network components in the work we have reviewed, and we aim to restore the most important components first so that their functions can be restored while repairs to the system are still ongoing.

III. Problem Definitions

On the occasion when an adversary information attack succeeds, the victim must have the capability to degrade gracefully and recover damaged data and/or services in real-time if it is to survive. It is necessary to immediately carry out damage assessment and recovery process in order to bring the systems to working states. Otherwise, the damage would spread to other unaffected systems that are interconnected. This happens when a valid user or an unaffected system module reads a damaged object during its computation and updates another object based on the compromised value, causing the latter damaged as well. As time goes on, more and more objects become affected in this manner causing the spread of damage to fan-out through the system quickly.

State-of-the-art assessment and recovery currently exist in types of critical infrastructure other than data systems. A review of threats that influence how critical infrastructures are protected is done in [22]. Among other threats, important ones that affect this research are cyber-attacks and cascading effects. A data system is especially vulnerable to the cascading effects of a cyber-attack, because of the nature of constant updates being made between different systems using data. In [21], a dynamic inoperability input-output model is introduced to determine how damage in one part of a system, such as power plants, water collectors, and transmitters can ultimately cascade down to end-users of the system. This method, as well as other similar methods in [23] and [24], primarily focus on physical threats to critical infrastructure. This research focuses on applying the same kind of resilience to data operations of critical infrastructures to protect them from digital threats.

For damage assessment and recovery purposes information about all processes that have been executed must be stored in the log (more on this presented later). This will help in determining the relationships among the processes, thus helping in establishing the damage trail. Moreover, during recovery, the operations of processes that have spread the damage have to be undone and then redone in order to produce the correct states of affected objects. The problems with existing systems are: (1) They do not store process execution information in the log, and they purge the log periodically, (2) their recovery mechanisms are not designed to undo the effects of executed processes, (3) the size of the log, as it must not be purged, will make it almost impossible to continue the recovery process in real-time, and (4) during the damage assessment and recovery process, the system remains unavailable to users. This delay induces a denial-of-service attack, which is highly undesirable in time-critical applications that the critical infrastructures are designed to provide. Due to the massive amount of data in the log that needs to be processed, the problem becomes even worse.

The goal of this research is to develop fast, accurate, and efficient damage assessment and recovery techniques so that critical information systems not only survive the attacks gracefully but will continue to operate providing as many vital services and functions as possible even before the system is fully recovered. In the next section, we explain how the relationships between different parts of critical systems can affect the spread of damage.

IV. Types of Damage Dependencies

Attacks can affect not only data systems, but also software that produces data. This happens when the software is maliciously changed to perform unintended actions. In this section, we discuss possible attack scenarios involving damaged data objects and software and how they can cascade into other systems.

A. Data to Data Damage

Data objects are frequently dependent on other data objects for computational updates. This makes their systems vulnerable to malicious attacks and cascading damage. Once a data object uses a damaged object to make a change, it becomes damaged too. This can happen to as many objects as the initially attacked objects have that are dependent on it. Figure 1 shows an example of cascading damage between data objects. If node C is targeted for an attack, then node E will become damaged due to its dependency on C. Then, node F and node G are also damaged because they are dependent on E.

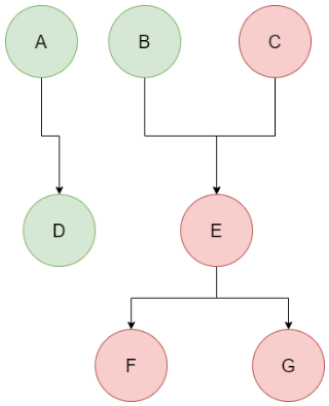


Figure 1: Cascading Damage from a Data Object

B. Software to Software Damage

If damaged software is used to influence computing done by another software, then the output of that software is also considered damaged. An example of a dependency between software is the use of libraries for applications. A damaged procedure within a library will result in damaged output for any software that calls that procedure. One difficulty with recovering from cascading damage from software to software is that it may be difficult to identify which cross-software calls used damaged procedures for computation. Consider a program *P* that uses two library functions *A* and *B*. *A* is an undamaged function and provides an undamaged output, while *B* is damaged and results in a damaged output. Even though we know the library is damaged, it is difficult to find which functions within the library are specifically damaged due to the library already being compiled into binary format. Therefore, we do not know if *A* or *B* is damaged, and after the library has been recovered, we must execute *P* entirely to ensure that we get the correct output, which causes recovery time to increase. After recovery, the resulting output from *A* is unchanged, while the output from *B* is corrected to the desired result. An example of software-to-software damage is shown in Figure 2. Even though the data is correct, a maliciously changed library can result in the software changing the output data.

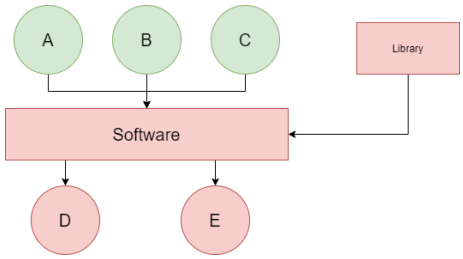


Figure 2: Cascading Damage from Software

C. Software to Data Damage

Any data changed by damaged software is also considered damaged. This is the simplest scenario involving damaged software because it is functionally the same as cascading damage between two data objects. After the software is repaired, any data that was damaged by the software must be computed again to complete recovery.

D. Data to Software Damage

Software that uses damaged data will not become damaged as a result. However, its output data will be considered damaged. This is because once a program is compiled, it cannot be changed by its input data. Therefore, if a software uses incorrect data to produce a damaged output, it is considered data-to-data damage instead. In Figure 3, a damaged node A is used as input for computation in a software. While the software itself is not damaged, its output nodes D and E become damaged.

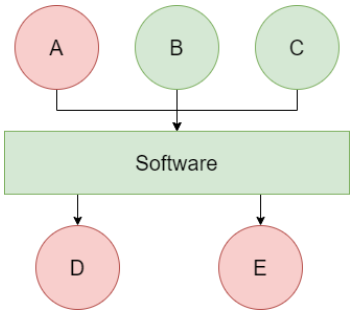


Figure 3: Cascading Damage from Data through Software

In the next section, we explain how our model can identify and recover damage optimally in detail. While it is

important to consider the possibility of damaged software affecting a system, our model focuses on the spread of damage within data objects. This is because damage can spread much faster through data objects than in software, and usually does so more frequently. Furthermore, the software can also be quickly fixed, while data objects are more numerous and may need many computations to get the desired output after repairs.

V. THE MODEL

In this section, we describe our model in detail. The first subsection defines important graphs and metrics that we use for our model. In the next subsection, we describe how the model is built and is used to determine an optimal recovery plan. Finally, we describe the algorithm we use to implement our model.

A. Definitions

We first define the concept of information flow in a system. This also defines dependencies among various objects in the system and is used in our graph-based model.

Definition 1: Given two objects O_i and O_j in a system, if the value of O_j is calculated using the value of O_i, we say that there is information flow from O_i to O_j. Thus, O_j is said to be dependent on O_i and is denoted as $O_i \rightarrow O_j$.

The above definition helps in determining the spread of damage in the system. That is, if an object is damaged, then all its dependent objects will be considered damaged. During recovery, the parent (pre-cursor) object must be recovered before any of its dependent objects can be recovered.

Next, we define a graph containing the set of objects and all possible paths among them. We call it Possible Paths graph and it spans the entire system of objects and all dependency paths among them. An example of this graph is shown in Figure 4(a).

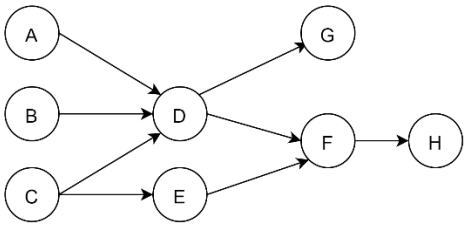


Figure 4a: The Possible Paths Graph (PPG)

Definition 2: Consider a system containing the set of objects O. The ***Possible Paths Graph*** (PPG) is built by having a node N_i for each object in O. There exists an edge E_{ij} from N_i to N_j in the PPG if there is a possibility that information may flow from N_i to N_j, that is, N_j may be modified based on the value of N_i.

The purpose of building a PPG is that it will help during the damage assessment preparation phase. By assuming the point of attack one can identify the set of items that may be affected consequently. Thus, security officers can be prepared for different types of eventualities.

The second set of objects contains the actual paths that were used to make changes in the system within a specified period, which for the purposes of the third graph that will be defined, is usually the time passed since an object has been damaged. This set is represented by the Active Paths Graph (APG), and all objects and dependencies in this set exist in the PPG. This graph will help in determining the damage flow in case of an attack. Given an initial attack point (an object), one can determine which objects in the system may be affected by the attack and which ones will not be. Therefore, the ability of the system to carry out its intended functions can be calculated. That is, during the recovery process, the set of damaged objects will be made unavailable while the rest can be made accessible. Knowing which objects will remain unaffected, one will be able to identify what services the system will be able to offer while the recovery continues.

Definition 3: The ***Active Paths Graph*** (APG) contains nodes N and edges E such that for every $N_i \in N$ and every $E_{ij} \in E$, both N_i and E_{ij} are also present in PPG, and E_{ij} illustrates an actual information flow; that is N_j was updated based on the value of N_i.

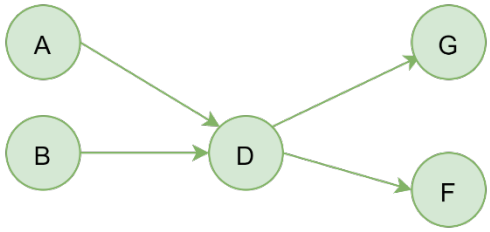


Figure 4b: The Active Paths Graph (APG)

Figure 4(b) provides an example of an Active Paths Graph and as can be seen, it is a sub-graph of Figure 4(a). As discussed before, once an initial attack point is determined, the APG will help in accurately determining the damage flow and the set of objects affected by the attack. As discussed before, as time goes on, more and more objects will be affected as new objects will be updated based on the value of an affected object. Thus, to stop the spread of damage, all affected objects must be quickly identified and taken offline as soon as possible. This can be achieved by doing a flow assessment using the APG. This leads to the concept of actual damage spread path showing exactly which objects were affected by an attack. If a system is damaged, we represent the spread of damage as the third set of objects, the Damage Spread Graph (DSG). The set of objects and dependencies in this graph must exist within the APG, as damage spread occurs when objects make changes based on their dependencies. Like how the APG is a subsection of the PPG, the DSG is a subsection of the APG. Figure 4(c) is an example of what a damage path may look like. It is important to note that over time, a damaged object will always cascade

its damage down to dependent nodes included in the APG. Definition 4 formally defines the DSG.

Definition 4: A ***Damage Spread Graph*** (DSG) contains nodes N and edges E such that for every $N_i \in N$ and every $E_{ij} \in E$, both N_i and E_{ij} are also present in APG and every node in N is damaged through an attack on the system. Moreover, an edge E_{ij} depicts that N_i was damaged first and then N_j was damaged through the flow of information from N_i to N_j.

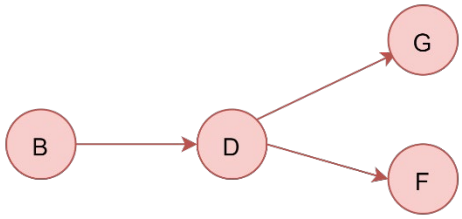


Figure 4c: The Damage Spread Graph (DSG)

Note that the edges between two objects may be bidirectional or recursive. For example, if an object O_j can have a dependency on object O_j and vice versa, then there will be a bidirectional edge between O_j and O_j. Similarly, if an object can be dependent on itself, it will result in a recursive graph. To clarify, let us consider an object "salary". When an employee receives an increment that is based on a percentage of the current salary of the employee, it causes the new salary to be dependent on the old salary and is depicted by using an edge from salary to salary itself. However, it must be noted that, for simplicity, we use neither bidirectional nor recursive edges in APG or DSG. Rather, when an object is modified, we note that as a new version of the object, thus creating a new node for the object with the version number.

To minimize the time needed to restore the most important objects within a system of object dependencies, we also define criteria used to determine the order in which repairs are made:

Definition 5: The ***criticality*** of a node N is its predetermined level of importance to the system's functions. This must be predetermined for the flexibility of the model to fit various systems and align the model with the goals of each specific system. For example, one system may need to prioritize certain components that other systems do not. The criticality of a component can be measured by various characteristics such as the intensity or scope of an impact caused by its failure as described in [14].

We assign a positive whole number to each node N to represent criticality. A lower assigned value indicates higher criticality. For example, a node N_i with a criticality of 2 would be considered more important than a node N_j with a criticality of 4. It is important to note that criticality values are not unique, meaning multiple nodes can have the same criticality value. When that happens, we use the following metric in the next definition to serve as a first "tiebreaker".

Definition 6: Objects that have more damaged dependencies take longer to repair. Therefore, the ***repair time*** of a node N is defined as how many inward-flowing edges E^i it is receiving damage from.

When two or more objects are assigned the same importance, we choose to first repair the one that has a lower repair time. For example, consider two nodes N_i and N_j that are equally critical. If N_i needs 5 other nodes repaired to repair it, and N_j needs 3 other nodes to repair it, then we will repair N_j first, because its operation can be restored more quickly than that of N_i.

Definition 7: The ***centrality*** of a node N is the number of outward-flowing edges E^o it has.

We use the above metric to decide the next object to repair when two or more are equal in both criticality and repair time. An object with a higher number of E^o will have higher centrality. Figure 5a and Figure 5b show two subsections of a DSG that highlights centrality. As shown in Figure 5a, N_4 has three nodes that are dependent on it: N_1, N_2, and N_3, while as Figure 5b depicts, N_6 only has a single node N_5 dependent on it. Assume that the repair algorithm has repaired the parent node(s) of N_4 and that of N_6. To clarify the situation, N_4 and N_6 need not have the same parents; it is just that both are in line to be repaired next. In this scenario, repairing N_4 before N_6 reduces the repair time for the three dependent nodes of N_4 instead of only one of N_6, which can make future repairs be performed faster. Therefore, N_4 is considered to have a higher centrality than N_6.

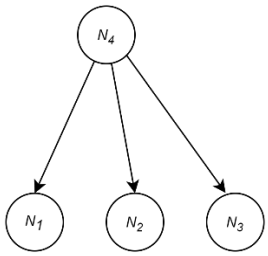


Figure 5a: A parent node with high centrality

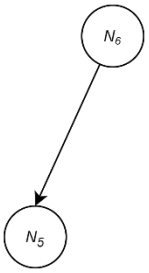


Figure 5b: A parent node with low centrality

Definition 8: The relationships between all nodes n in a data system can be expressed by an ***adjacency matrix*** A, where each element represents an edge $e_{i,j}$ between nodes n_i and n_j such that the parent node is given by the element's row and the child node is given by the element's column. The value of each element is binary – if an edge going from one node to

another exists, then the value of its respective element is 1, otherwise, it is 0. Therefore, for each element $a_{i,j}$ in A:

$$a_{i,j} = \begin{cases} 1 \text{ if } (i,j) \in N_i \\ 0 \text{ if } (i,j) \notin N_i \end{cases}. \quad (1)$$

Figure 6 depicts the PPG shown in Figure 4a. as an adjacency matrix. Each cell with a value of 1 is an existing edge on the graph. For example, node D has 3 parent nodes and 2 child nodes. The edges coming from nodes A, B, and C are highlighted on D's row. Likewise, its outgoing edges toward nodes F and G are highlighted on D's column. All node relationships are translated this way, so as there are 8 edges in the graph, there is also 8 highlighted elements in Figure 6.

Since an adjacency matrix can be used to represent the relationships between data nodes in a uniform manner, it is used for our implementation of a damage assessment algorithm, which will be covered in the next two subsections.

	A	B	C	D	E	F	G	H
A	0	0	0	0	0	0	0	0
B	0	0	0	0	0	0	0	0
C	0	0	0	0	0	0	0	0
D	1	1	1	0	0	0	0	0
E	0	0	1	0	0	0	0	0
F	0	0	0	1	1	0	0	0
G	0	0	0	1	0	0	0	0
H	0	0	0	0	0	1	0	0

Figure 6: An adjacency matrix of Figure 4a.

B. Model Description

The model uses the three graphs defined in the previous section to construct a representation of a given system and its sustained damage from the time of the initial attack. The PPG is a preprocessed map of all components and dependency paths within a system. We assume that we know how much time has passed since the initial attack and build the APG by including components and dependency paths that were used in a transaction log in that period. By knowing the component where the initial attack occurred, we build the DSG by tracing the damage through the transaction log. For damage to spread from one component to the next, it must follow two criteria: 1) there is a damage node N_i that has an edge E_{ij} flowing from it to node N_j and 2) E_{ij} is used for a transaction while N_i is damaged. For the DSG to exist, the initial attack must occur within the APG, otherwise there is no cascading damage.

The goal of the model is to find the optimal sequence of repairs to restore the most important operations of a system as quickly as possible. We use the metrics defined in the previous section to decide which components should be repaired first. The first metric is criticality – the most critical components must be restored first to resume important operations. However, these components may also be dependent on other components that are damaged. These components must be repaired first before the base component can be repaired. At this point, the same problem is applied to the dependency components, and the most critical one is chosen first. If there is a tie, then components with a lower repair time are picked first. For example, a component that has two damaged parent components will be prioritized over a component with three or more damaged parent components if both components are equally critical.

To clarify, let us consider the graph presented in Figure 7. As shown in the figure, nodes N_1, N_2, and N_3 are dependent on N_4. Assume that the damage assessment method identified N_4 as damaged; thus, nodes N_1, N_2, and N_3 are also identified as damaged. During the recovery process, N_4 was recovered before the other three nodes. However, since it has three dependents all of which are damaged, the question is, which one should be repaired first. As our goal is to have the vital functions of the system to be made available before the other operations, our algorithm would choose the node among N_1, N_2, and N_3 having the most criticality.

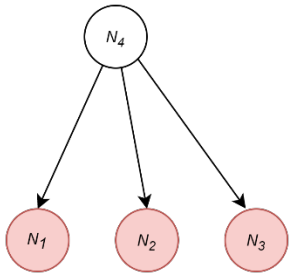


Figure 7: Recovery sequence decision

Repair time is not affected by how the parent components are ordered. For example, consider two scenarios with nodes N_1, N_2, N_3, and N_4. In the first scenario, node N_1 is dependent on nodes N_2, N_3, and N_4. In the second scenario, node N_1 is dependent on node N_2, node N_2 is dependent on node N_3, and node $N3$ is dependent on node N_4. In both scenarios, nodes N_4, N_3, and N_2 must all be repaired before node N_1 can be repaired, so N_1 will always have the same repair time. In short, repair time can simply be considered as the number of upstream components. Similarly, the third metric, centrality, can be considered as the number of downstream components. This is the third metric used to determine repair order in case multiple components are equal in criticality and repair time. This metric is not prioritized over the first two because it does not directly contribute toward the stated goals of our model, but it can optimize future repairs by lowering the repair time of more components than repairing other components would.

C. The Algorithm

First, we discuss the primary objective of our work. Let us consider the notations used in Table I.

TABLE I. NOTATIONS

Notations	Descriptions
$P = (V, E)$	Possible Path Graph
$A = (V_A, E_A)$	Active Path Graph ($V_A \subseteq V, E_A \subseteq E$)
$D = (V_D, E_D)$	Damage Spread Graph ($V_D \subseteq V_A, E_D \subseteq E_A$)
$D = (V_C, E_C)$	Critical Node Graph ($V_C \subseteq V_D, E_C \subseteq E_D$)
δ_{ij}	Decision to fix edge i to j
δ_i	Decision to fix node i
t_i	Time to fix node i
c_i	Centrality of node i
P_{ij}	Dependency indicator of node i and j

Our objective is to find $\min \sum_{i \in V_D} t_i \delta_i$ subject to

$$\delta_i \sum_{j \in V_C} P_{ij} \leq \sum_{j \in V_C} P_{ij} \delta_j \quad \forall i, j \in V_C \qquad (2)$$

$$\delta_i c_i \geq \sum_{(i,j) \in E_C} \delta_{ij} \quad \forall i \in V_C \qquad (3)$$

$$P_{ij} \in \{0,1\} \quad \forall i, j \in V_C \qquad (4)$$

$$\delta_i, \delta_{ij} \in \{0,1\} \quad \forall i \in V_C, (i,j) \in E_C \qquad (5)$$

That is, the goal is to minimize the time required to fix all critical nodes subjected to conditional constraints of the system. To make sure that each preceding nodes of i are fixed before node i being processed, condition (2) is used. For example, if there is a node j connecting to i but in a prequel order, the sum product of all nodes j status and dependency indicator P_{ij} should be greater or equal than the product of sum of all dependency indicator P_{ij} with node i. To make sure that there would not be more out-going flows than the given capability of node i, equation (3) is imposed to make sure the total out-going edge would not surpass the centrality of node i. Conditions (4) and (5) were built to impose the binary attribute of the dependency indicator P_{ij}, the decision whether to fix node i or edge from node i to node j.

The algorithms provided in this section use the model described in the previous section to compute the optimal order of repairs to restore the most important functions of a system first. When an attack occurs, we expect an Intrusion Detection System (IDS) to identify the attack and provide the initial point of damage. The working principles of IDSs are not within the scope of this work and so, not described here.

Algorithm 1 creates the PPG from the system's data. The PPG must record all possible dependency paths, including those that may not have been used by the system yet. Therefore, it is necessary to consult the system's designers so the algorithm can be provided full information on the system's data objects to fully construct the PPG. As stated in Definition 8, the PPG, APG, and DSG are all represented by a binary adjacency matrix, where child nodes and parent nodes are represented by the columns and rows, respectively. Each element in one such matrix is the edge between the given parent and child node, with a value of 1 indicating that the edge does exist in the graph and a value of 0 indicating that it does not exist.

Algorithm 1: Initializing the Possible Paths Graph
Result: Adjacency matrix M^{PPG} representing the PPG
1 for each object O_x and O_y in the system
1.1 if edge E_{xy}can exist
1.2 $M^{\text{PPG}}(x, y) = 1$
2 Return M^{PPG}

After receiving notification from an IDS, a precise damage assessment is performed. If the damage assessment process is unable to make accurate assessment, i.e., in case a damaged node is not correctly identified, it and its dependent nodes, which are also damaged, will remain unrecovered. This will result in valid users or procedures reading them and spreading damage by updating other objects, as discussed earlier. For a detailed discussion on damage assessment, one may review [9] and [10], which were developed particularly for database systems. However, the methods are still applicable to critical infrastructure systems. Below we provide a basic mechanism to carry out the assessment.

Damage assessment begins with the APG, which shows the actual dependency relationships among the objects in the system (Note that the APG can be built as transactions are executed and dependencies are established among various nodes of the PPG). Given the initial attack point, the corresponding node is then marked as damaged. This is the starting node of the DSG. Then by scanning the log from the corresponding location of the attack point, transactions that read the marked node are identified. Any objects written by those transactions are then marked as damaged in the APG. This process continues until the end of the log. Finally, all unmarked nodes and the edges showing their dependencies are removed. The resulting graph is the completed DSG.

The APG is constructed in Algorithm 2. This algorithm reads the transaction log and creates a node each time a new data object is mentioned in the log. The object's dependencies are depicted as the node's edges in the APG. When an existing data object is updated, it becomes a new object in the transaction log. As described earlier, this is done to prevent recursive dependencies in the APG.

Once damage assessment is carried out, recovery procedure must begin immediately in order to make the system operational quickly. We use Algorithm 3.1 as the main procedure to initialize an object set for repairs. The algorithm starts by initializing the set of damaged objects *O*. Each node *N* within *O* consists of a system component and its relationships with other nodes in *O*. As mentioned previously under Definition 4, some system components may have recursive or bidirectional dependencies between each other. Therefore, system components can have repeat nodes within *O* to represent their different versions. Each node is assigned values for criticality, repair time, and centrality. Using those metrics, the algorithm determines an initial target node N_0 based on criticality. If there are two or more nodes with the highest criticality, then the node with the lower repair time is selected. In the event of another tie, the node with higher centrality is selected. Further ties are broken by random

selection. N_0, along with O and the repair queue Q, are used to make the first call to the recursive function Algorithm 3.2 at step 4.5. Algorithm 3.1 proceeds until O is completely empty, and then the repair queue is finalized, and Q is printed.

As previously discussed, a node must have its parent nodes repaired before it can be considered eligible for repairs. Algorithm 3.2 ensures that nodes are scheduled for repairs in the proper order while still adhering to the rules set for determining priority. It does this by using a while loop to check the currently selected node N for repair eligibility. If N is eligible for repairs, then it is removed from O and Q is updated, then returned. If N is not eligible, then O', a subsection of O made up of all dependency paths above the currently selected node is created and used to find the next highest priority node N' within O'. Algorithm 3.2 is recursively called using N' and O', which can either result in the node's repair or another node being selected for repair again. The recursive nature of this algorithm ensures that each time a decision needs to be made on which node needs to be repaired next, it will prioritize criticality and efficiency among all the nodes that can be repaired at any given step. In this way, the bulk of the work done by the algorithm is choosing the next object for repair within each iteration. Each function call will result in one object being repaired and $n-1$ additional function calls, where n is the number of nodes within the set of nodes being passed. Since repaired objects need to be removed from the DSG, function calls will need to update and return the global DSG and Q.

Algorithm 2: Initializing the Actual Paths Graph
Result: Adjacency matrix M^{APG} representing the APG
Parameters: transaction log T, containing the set of edges E that were used to make updates
1 For each edge E_{xy} in T
  1.1 $M^{\mathrm{APG}}(x, y) = 1$
2 Return M^{APG}

Algorithm 3.1: Initialization for object set repair
Result: Queue of objects ordered by repair priority
1 Initialize set of damaged objects O
2 Preprocess object priority using criticality, repair time, and centrality
3 Initialize repair queue Q
4 while *O has damaged nodes remaining*
  4.1 Select the highest critical node(s) N within O
  4.2 if *Two or more nodes are tied for highest criticality*
    4.2.1 Select the node(s) N with the lowest repair time R within O
  4.3 if *Two or more nodes are tied for lowest repair time*
    4.3.1 Select the node(s) N with the highest centrality within O
  4.4 if *Two or mode nodes are tied for highest centrality*
    4.4.1 Select a single node at random from those still tied
  4.5 Update repair queue(N_0, O, Q) → Q
5 Print Q

Algorithm 3.2: Recursive repair function
Result: Schedules a node N for repairs and returns the updated repair queue Q
1 Update repair queue(*Selected node N, object set O, repair queue Q)*:
2 while *Current object has unrepaired dependencies*:
  2.1 Create subset of damaged nodes O' of all nodes N' and edges E' that N is dependent on
  2.2 Select the highest critical node(s) N' within O'
  2.3 if *Two or more nodes are tied for highest criticality*
    2.3.1 Select the node(s) N' with the lowest repair time R within O
  2.4 if *Two or more nodes are tied for lowest repair time*
    2.4.1 Select the node(s) N' with the highest centrality within O
  2.5 if *Two or mode nodes are tied for highest centrality*
    2.5.1 Select a single node at random from those still tied
  2.6 Update repair queue(N'_0, O', Q) → Q
  2.7 Remove the most recent object in repair queue from O
3 Repair N
4 Add N to Q
5 Return Q

The algorithm produces a list of system nodes in the order in which they should be repaired. Recovery procedure then continues to the next step to begin repairs on the system. It is important to note that while repairs are simulated by the algorithm, the process for repairing the actual components of the system is not within the scope of this work.

VI. Experiments and Analysis

In this section we present the results from simulations done using our model. We first describe the setup used for our experiment, then present the results from each simulation that we ran and explain the implications of the results.

A. Experiment Description

We evaluated the efficiency of our method by using it to find the number of nodes needed to repair every critical node in a DSG simulation with various parameters. The simulation constructed a DSG with randomly assigned relationships between nodes. The parameters used to build each DSG were total nodes, percentage of critical nodes, maximum number of parent nodes per node, and maximum number of children nodes per node. After the DSG was built, the set of critical nodes was randomly picked from the entire DSG for our method to compute the repairs required to bring all critical nodes back into a good state. For each simulation, 25 different sets of critical nodes were tested, and the average number of required repairs was reported by the simulation. We ran four experiments to test the changes to the number of

required repairs caused by changing each parameter. For every experiment, we set the control variables to be 10,000 total nodes, 5% critical nodes, and a maximum of 6 parent nodes and 6 child nodes per node. We compared our method to modified versions of two common traversal algorithms: breadth-first search (BFS) and depth-first search (DFS). The BFS algorithm repaired nodes starting with all the root nodes, then all the children of the root nodes, and then the next set of child nodes until it reached the bottom. On the other hand, DFS repairs a single parent node, then repairs one of its child nodes, and repeats that process until it reaches a node with no children. When that happens or if it has already repaired all child nodes for a given node, it goes back up to the previous node and repairs another child node. The exception to this process is if a node has other parent nodes that have not been repaired yet. The algorithm will break the traditional BFS or DFS order to ensure that the node is eligible for repair. We include the results for the BFS and DFS repair algorithms with our own method.

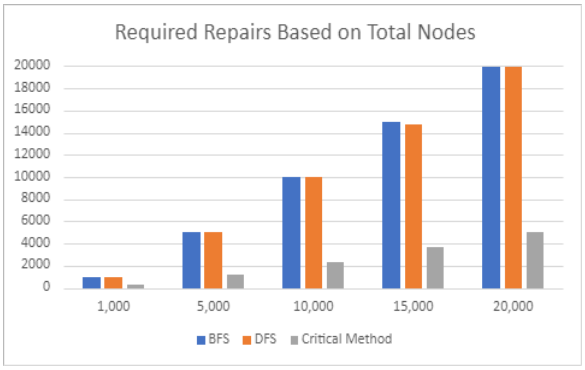


Figure 8a: Required repairs based on Total Nodes

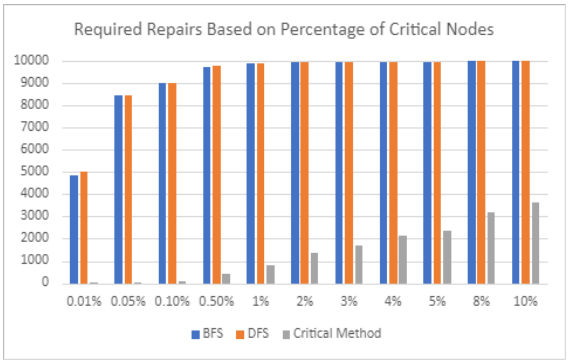


Figure 8b: Required repairs based on Critical Node Percentage

B. Results and Analysis

We present two graphs for each round of simulations in Figures 8 and 9. The former figure is a bar graph highlighting the different in required repairs between our algorithm and BFS and DFS traversals. Figure 9 shows the changes in the percentage of nodes repaired between each parameter of a simulation. Our results for required repairs based on total nodes is shown in Figure 8. It shows that the required repairs maintain a constant ratio between them and the total nodes with the given control variables, as each experiment resulted in a little less than 25% of the total nodes needing to be repaired. Figure 8b shows the results for required repairs needed for different percentages of critical nodes. We found that while the number of required repairs increases as the percentage of critical nodes increases, less additional repairs are needed when the percentage is higher. Therefore, a high number of critical nodes require less repairs per critical node than a lower number of critical nodes. Figures 8c and 8d show the change in required repairs for different maximums of children and parent nodes per node respectively. We found that with a low number of maximum children per node, more repairs are required. This is because the system of nodes becomes narrower in shape. In contrast, when there is a higher maximum of parent nodes per node, the required repairs see an increase. This indicates that a critical node is more likely to have additional parent nodes, which would require more repairs than a critical node with fewer parent nodes. Across all graphs, we can see that targeting specific nodes for repair drastically shortens the time needed to recover critical functions, as the average required repairs for the BFS and DFS algorithm in most simulations was slightly less than the total number of nodes in the simulation.

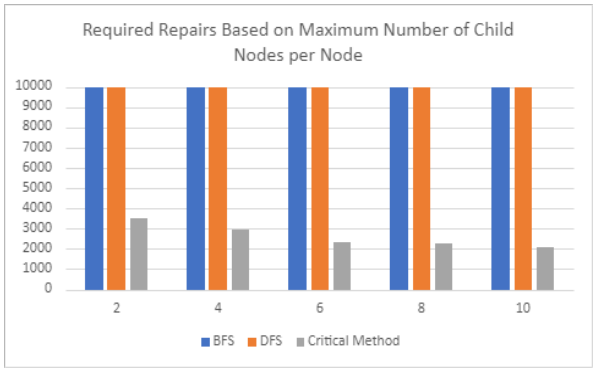


Figure 8c: Required repairs based on Maximum Number of Child Nodes

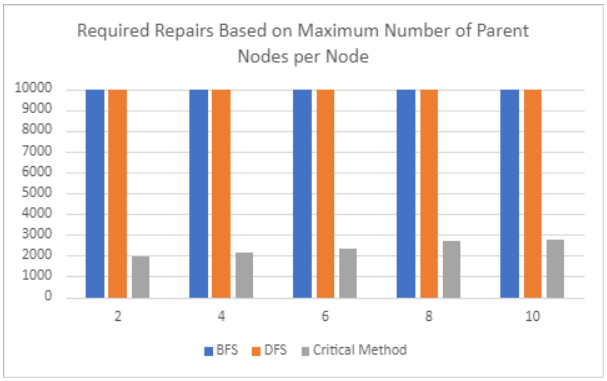


Figure 8d: Required repairs based on Maximum Number of Parent Nodes

C. On BFS and DFS algorithms

Before running our simulations, we expected our algorithm to outperform BFS and DFS searches due to it specifically targeting critical nodes for repair first. However, these two algorithms ended up traversing nearly the entire system before repairing all critical nodes. The best way to explain this is to consider the order by which each algorithm repairs nodes as a static ordered list. Our algorithm sorts this list after nodes are assigned criticality, so the final critical node that needs to be repaired ends up being close to the front of this list. On the other hand, BFS and DFS do not consider criticality when sorting this list, so each node's criticality on their lists after sorting is effectively random. What determines how many nodes need to be repaired is ultimately the final critical node in the list, so the odds of this node not being in the final thousand or even hundred nodes are quite low. Therefore, BFS and DFS will need to repair almost all the nodes except for when the percentage of critical nodes is lower. When that happens, it is more likely that the final critical node will be closer to the middle of the repair order instead.

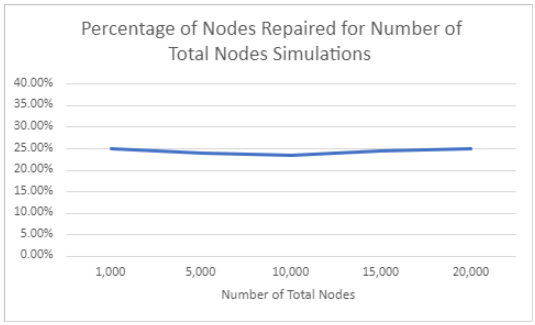


Figure 9a: Changes in required repairs based on maximum number of parent nodes

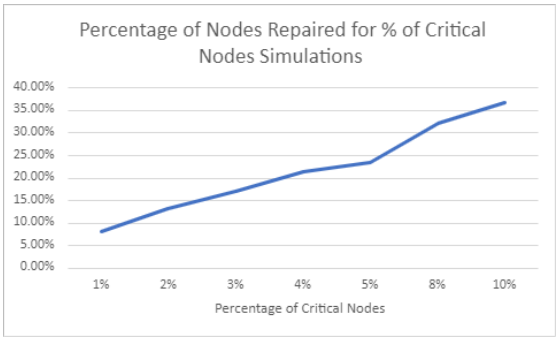


Figure 9b: Changes in required repairs based on the percentage of critical nodes

VII. Conclusion

In this research, we have presented a method to repair data objects that prioritizes quick recovery for the most important components of a system. This allows for the partial restoration of functions during the recovery process with an emphasis on restoring service to the most necessary functions. This was first done by building out three graphs to represent the entire system, what changes the system made after an attack, and the cascading damage as a result of those changes. Next, we developed an algorithm to optimally schedule repairs by using those graphs to find damage paths that affect the most critical nodes of a system and calculate the fastest repair order to fully restore those nodes. Lastly, we presented results from a simulation of our algorithm and the effects on changing the parameters of the damage paths. Our work is most applicable to protecting critical infrastructure systems where services need to be restored as quickly as possible to avoid economic or societal disruptions.

Further work includes considering the frequency at which an object is used to update its dependencies. Objects that are updated at a higher frequency would be prioritized as more important. Additionally, a method to select the order of repairs for non-critical objects after all critical objects have been repaired is also needed.

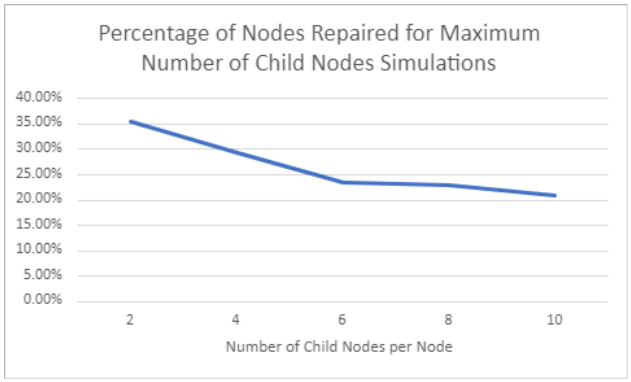


Figure 9c: Changes in required repairs based on the maximum number of child nodes per node

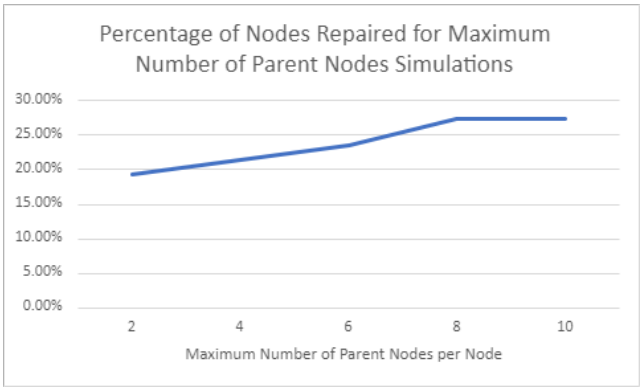


Figure 9d: Changes in required repairs based on the maximum number of parent nodes per node

Acknowledgement

This work has been supported in part by grant H98230-20-1-0419 issued by the National Security Agency as part of the National Centers of Academic Excellence in Cybersecurity's mission to expand cybersecurity research and education for the Nation.

REFERENCES

[1] J. Burns, B. Panda, T. Bui, "Modeling Damage Paths and Repairing Objects in Critical Infrastructure Systems", The Fifteenth International Conference on Emerging Security Information, Systems and Technologies SECUREWARE 2021, Athens, Greece, pp. 88-93, Nov. 14-18, 2021, ISBN 978-1-61208-919-5

[2] E. Viganò, M. Loi. and E. Yaghmaei, "Cybersecurity of Critical Infrastructure", In Christen M., Gordijn B., Loi M. (eds), The Ethics of Cybersecurity, The International Library of Ethics, Law and Technology, vol. 21, SpringerLink.

[3] *Critical Infrastructure Security*: https://www.dhs.gov/topic/critical-infrastructure-security. [retrieved: October 2021]

[4] D. E. Sanger and N. Perlroth, (2021, May 14). "Pipeline Attack Yields Urgent Lessons about U.S. Cybersecurity", https://www.nytimes.com/2021/05/14/us/politics/pipeline-hack.html. [retrieved: October 2021]

[5] A. Anastasios, "Is the Electric Grid Ready to Respond to Increased Cyber Threats?", https://www.tripwire.com/state-of-security/ics-security/electric-grid-ready-increased-cyber-threats/. [retrieved: October 2021]

[6] B. Barrett, "An Unprecedented Cyberattack Hit US Power Utilities", https://www.wired.com/story/power-grid-cyberattack-facebook-phone-numbers-security-news/. [retrieved: October 2021]

[7] K. O'Flaherty, "U.S. Government Issues Powerful Cyberattack Warning as Gas Pipeline Forced into Two Day Shut Down" https://www.forbes.com/sites/kateoflahertyuk/2020/02/19/us-government-issues-powerful-cyberattack-warning-as-gas-pipeline-forced-into-two-day-shut-down/#5f3061645a95. [retrieved: October 2021]

[8] M. Lewis, "Cyberattack Forces Gas Pipeline Shutdown", https://www.jdsupra.com/legalnews/cyberattack-forces-gas-pipeline-shutdown-76217/ [retrieved: October 2021]

[9] B. Panda and J. Giordano, "Reconstructing the Database after Electronic Attacks. In: Jajodia S. (eds) Database Security XII. IFIP — The International Federation for Information Processing, vol. 14. Springer, Boston, MA, 1999.

[10] S. Patnaik and B. Panda, "Transaction-Relationship Oriented Log Division for Data Recovery from Information Attacks" Journal of Database Management, vol. 14. No. 2. pp. 27-41, 2003.

[11] P. Kotzanikolaou, M. Theoharidou, and D. Gritzalis, "Cascading Effects of Common-Cause Failures in Critical Infrastructures". In: J. Butts and S. Shenoi (eds) Critical Infrastructure Protection VII. IFIP Advances in Information and Communication Technology, vol. 417, 2013. Springer, Berlin, Heidelberg.

[12] J. Ding, Y. Atif, S. Andler, B. Lindström, and M. Jeusfeld, "CPS-based Threat Modeling for Critical Infrastructure Protection". ACM SIGMETRICS Performance Evaluation Review. 45. pp. 129-132, 2017. 10.1145/3152042.3152080.

[13] E. Canzani, H. Kaufmann, and U. Lechner, "Characterising Disruptive Events to Model Cascade Failures in Critical Infrastructures", The Fourth International Symposium for ICS & SCADA Cyber Security Research, Belfast, Northern Ireland, 2016.

[14] D. Rehak, J. Markuci, M. Hromada, and K. Barcova, "Quantitative Evaluation of the Synergistic Effects of Failures in a Critical Infrastructure System", International Journal of Critical Infrastructure Protection, vol. 14, pp. 3-17, 2016. ISSN 1874-5482

[15] N. Bartolini, S. Ciavarella, T. F. La Porta, and S. Silvestri, "Network Recovery after Massive Failures," 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), pp. 97-108, 2016.

[16] S. Ciavarella, N. Bartolini, H. Khamfroush, and T. Porta, (2017). "Progressive Damage Assessment and Network Recovery after Massive Failures," IEEE INFOCOM 2017 – IEEE Conference on Computer Communications, pp. 1-9, 2017.

[17] J. Štoller and P. Dvořák, "Basic Types of Critical Infrastructure Objects," 2019 International Conference on Military Technologies (ICMT), pp. 1-7, 2019. doi: 10.1109/MILTECHS.2019.8870031.

[18] D. Rehak, P. Senovsky, and M. Hromada, "Analysis of Critical infrastructure Network. Z. Chan, m. Dehmer, F. Emmertsteib, and Y. Shi, *Modern and interdisciplinary* problems in network science: a translation research perspective. 1. Boca Raton, FL, USA: CRC Press, pp. 143-171, 2018. ISBN 978-0-8153-7658-3.

[19] M. Luskova, M. Titko, and B. Leitner, "Multilevel Approach to Measuring Societal Vulnerability due to Failure of Critical Land Transport Infrastructure", The World Multi-Conference on Systematics, Cybernetics and Informatics, Miami, FL, July 2016.

[20] B. Genge, P. Haller, I. Kiss, "A Framework for Designing Resilient Distributed Intrusion Detection Systems for Critical Infrastructures", International Journal of Critical Infrastructure Protection, vol. 15, pp. 3-11, 2016. ISSN 1874-5482, https://doi.org/10.1016/j.ijcip.2016.06.003.

[21] X. He, E. Cha, "Modeling the Damage and Recovery of Interdependent Critical Infrastructure Systems from Natural Hazards", Reliability Engineering & System Safety, vol. 177, pp. 162-175, 2018. ISSN 0951-8320, https://doi.org/10.1016/j.ress.2018.04.029.

[22] R. Osei-Kyei, V. Tam, M. Ma, F. Mashiri, "Critical Review of the Threats Affecting the Building of Critical Infrastructure Resilience", International Journal of Disaster Risk Reduction, vol. 60, 2021, ISSN 2212-4209, https://doi.org/10.1016/j.ijdrr.2021.102316.

[23] Y. Fang and G. Sansavini, "Optimum Post-disruption Restoration under Uncertainty for Enhancing Critical Infrastructure Resilience", Reliability Engineering & System Safety, vol. 185, pp. 1-11, 2019. ISSN 0951-8320, https://doi.org/10.1016/j.ress.2018.12.002.

[24] M. Xu, M. Ouyang, Z. Mao, X. Xu, "Improving Repair Sequence Scheduling Methods for Post-disaster Critical Infrastructure Systems", Computer Aided Civil and Infrastructure Engineering, vil. 34, pp. 506–522, 2019. https://doi.org/10.1111/mice.12435

Implementation of a Software Based Glitching Detection Mechanism

Jakob Löw
Technische Hochschule Ingolstadt
Ingolstadt, Germany
jakob.loew@thi.de

Dominik Bayerl
Technische Hochschule Ingolstadt
Ingolstadt, Germany
dominik.bayerl@carissma.eu

Hans-Joachim Hof
Technische Hochschule Ingolstadt
Ingolstadt, Germany
hof@thi.de

***Abstract*—Clock glitching is an attack surface of many microprocessors. While fault resistant processors exist, they usually come with a higher price tag, resulting in their cheaper alternatives being used for small embedded devices. After describing the effects of fault attacks and their application to modern microprocessors, this paper presents the concept and implementation of a novel software based approach at protecting programs from fault attacks. Even though the protection mechanism can be automatically added to a given program in a special compiler step, its use case is not to protect the full program. As shown by the performance analysis in this paper, the approach comes with heavy performance implications, making it only useful for protecting important parts of programs, such as initialization, key exchanges or other cryptographic implementations.**

***Index Terms*—computer security; clocks; microcontrollers; program compilers; program control structures**

I. Introduction

Hardening software against glitching attacks manually is a tedious task and requires a trained developer. Hardware based glitch detection on the other hand increases cost of production. Thus the most efficient approach in order to protect against glitch attacks is with generalized and automated software mechanisms. The goal of this paper is to introduce and rate a software based approach, protecting a program from clock glitching attacks.
In order to introduce this approach, first, the nature and effects of glitching attacks in general and clock glitching attacks in particular are described in Section II. Section III discusses state of the art software based protection mechanisms. Then an approach at detecting glitch attacks is introduced in Section IV. This paper is the extended version of Löw et al. [1], it introduces an implementation of the glitch detection approach in Section V. The performance impact of the approach is then rated using this implementation in Section VI.

II. Glitching Attack Models

In embedded IT Security, glitching attacks are a special kind of side channel attacks. Their target is to trigger misbehaviours of the target processor in order to alter execution or data flow. A typical goal of a glitch attack is changing the execution flow, such that one instruction is skipped. For example, when glitching the conditional branch instruction of a signature check, the check is skipped and the program continues even if the signatures did not match. Triggering a glitch while the processor is loading a value from memory can cause the memory load to not finish correctly and often results in a zero value being loaded instead. Thus, glitching the data flow is often used to attack cryptographic algorithms by glitching the load of keys from memory or by glitching arithmetic operations [3].
The next Subsection will first describe clock glitching attacks, which this paper focuses on, in detail. Afterwards Section II-B will cover the exact effects of clock glitches targeting microprocessors using Atmels AVR microarchitecture.

A. Clock Glitching

Clock glitching is a specific form of glitching attacks. A glitch in the target processor is triggered by altering the provided clock signal. Normally a clock signal is generated by an oscillator with a constant frequency; Rising that frequency is called overclocking. Each processor has a maximum operating frequency, if the clock frequency rises above this threshold the processor starts to behave abnormally.
In a classical clock glitching attack, only a single targeted glitch is inserted into the clock signal, i.e., a second high signal is inserted causing the current instruction to not complete before the next one starts its execution. The effects depend on various parameters as well as on the processors architecture and design.
Figure 1 shows the electrical potential of a clock line during a clock glitch attack. The first Section, labeled as cycle A, shows a regular clock cycle, while cycle B shows a clock cycle with a glitch inserted [6].

B. Effects of Clock Glitches on AVR Microprocessors

The research by Balasch et al. [6] goes into detail about what exactly happens when a microprocessor is attacked by a glitching attack. They used a Field Programmable Gate Array (FPGA) to generate a clock signal for ATMega163 based smart cards. The FPGA allows clock signal modifications, such as inserting a glitch at a specific location. The ATMega runs a special firmware, which places all registers in a known state, executes the instruction targeted by the glitch and then examines the state of all registers of the microprocessors. From the transformations between the start state and the result state the executed instruction can be derived. This, however, is a non trivial task. For example, when before the instruction the value

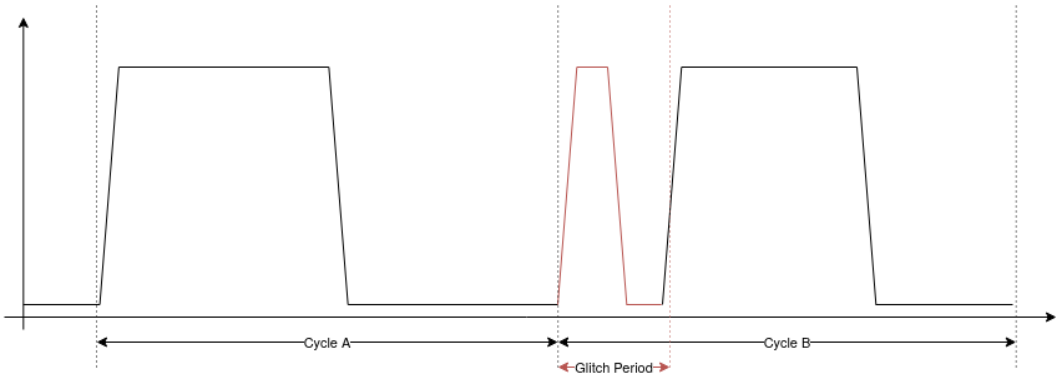


Fig. 1: Injection of a Clock Glitch

`0x0f` was in register `r18` which changed to `0xf0` afterwards the executed instruction could either be a 4-bit left shift or an addition with `0x51`. Multiple runs with the same glitch period, the same instruction but different input states have to be performed in order to be able to identify the actual executed instruction.

With these methods Balasch et al. [6] show the actual effect of clock glitches with different glitch periods on a target instruction. During instruction fetching the value of the instruction to execute next changes from the previous instruction to zero and then to the value of the following instruction. By injecting a glitch into this transition, depending on the length of the glitch period, either a decayed version of the previous instruction or a decayed, i.e., not yet fully loaded, version of the current instruction can be executed. Figure 2 shows this behaviour for a *Set all Bits in Register* (`SER`) instruction followed by a *Branch if Equal* (`BREQ`) instruction. In this specific case, for a glitch period up to 28 ns a decayed version of the `BREQ` instruction is executed. From 32ns and upwards an intermediate value of the transition from zero to `SER` is executed [6].

Glitch period	Instruction	Opcode (base 2)
	`TST R12`	`0010 0000 1100 1100`
-	`BREQ PC+0x02`	`1111 0000 0000 1001`
	`SER R26`	`1110 1111 1010 1111`
$\leq$ 57ns	`LDI R26,0xEF`	`1110 1110 1010 1111`
$\leq$ 56ns	`LDI R26,0xCF`	`1110 1100 1010 1111`
$\leq$ 52ns	`LDI R26,0x0F`	`1110 0000 1010 1111`
$\leq$ 45ns	`LDI R16,0x09`	`1110 0000 0000 1001`
$\leq$ 32ns	`LD R0,Y+0x01`	`1000 0000 0000 1001`
$\leq$ 28ns	`LD R0,Y`	`1000 0000 0000 1000`
$\leq$ 27ns	`LDI R16,0x09`	`1110 0000 0000 1001`
$\leq$ 15ns	`BREQ PC+0x02`	`1111 0000 0000 1001`

Fig. 2: Instruction decay based on glitch period

C. Glock glitching with modern Microprocessors

As microcontrollers such as the AVR tiny and mega series usually clock between 1 and 8 MHz they tend to have a direct clock input line, where a oscillator with the desired clock frequency has to be attached.

On higher clocking microcontrollers such as the *ARM Cortex M0* series operating on a clock frequency of usually around 50MHz to 100MHz a phase locked loop (PLL) is used to generate the clock signal [11]. A PLL uses an input signal of an oscillator and is able to multiply that signal allowing it to generate higher frequencies [4]. This is achieved by dividing the output of a voltage controlled oscillator (VCO) and synchronizing this divided voltage with an input voltage obtained from a quartz oscillator. This way the input frequency is effectively multiplied achieving a higher frequency.

A PLL output does not immediately respond to changes in the input frequency, which means a classical clock glitch has little to no effect on the actual clock frequency of the microprocessor. Increasing the input frequency for a longer period of time, does however make the PLL output frequency rise. This effect is used by B. Selmke et al. in [11] to induce glitches into an *ARM Cortex M0* processor even though it utilizes a PLL based clock multiplier. Figure 3 shows the principle of this approach: By increasingt he input frequency for a specific duration the processor frequency generated by the PLL increases. While the processor might be able to be slightly overclocked over the nominal frequency, at a specific point the frequency becomes too high for the processor to work correctly, at which point the processor shows faulty behaviour. The frequency required for entering the CPU fault zone depends on the chip kind, quality and even on the instructions executed. This makes glitching processors with PLLs hard, but [11] proves it is still achievable by attacking an AES implementation running on an *ARM Cortex M0*.

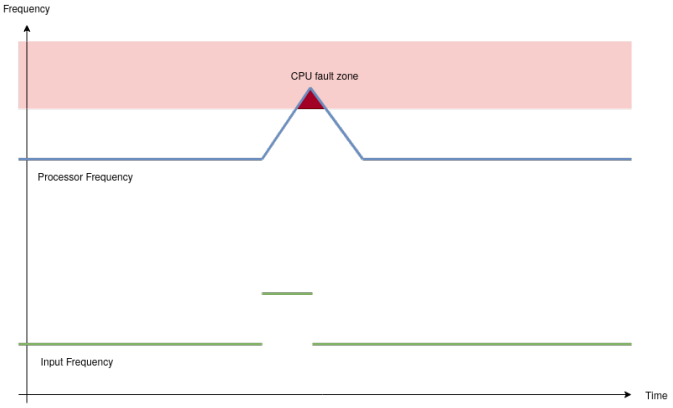


Fig. 3: Glitching a processor with a PLL

III. Existing Software based Glitch Detection Techniques

Papers covering fault based attacks on cryptographic implementations date back to 1997, and there are already multiple papers covering protection mechanisms against fault attacks using software or hardware based countermeasures. The software based countermeasures are usually based on either duplicating instructions or validating computations. The following sections describe some of the common approaches at glitch detection by example, before a novel approach is discussed in Section IV.

A. Instruction duplication mechanisms

A very common approach at protecting code from glitch attacks is instruction duplication or even triplication. It is usually implemented at a very late stage in the compilation process and works by simply duplicating memory load or even arithmetic instructions and checking their results for equality. A simple ARM64 assembly example is shown in Figure 4. Instead of only loading the value at `x0` once into register `w0` it is loaded a second time into `w1`. If a glitch occured in one of the two instructions, i.e., a wrong value was read from memory, the comparison check fails and an error handler is called.

```
ldr     w1, [x0]
ldr     w0, [x0]
cmp     w1, w0
bne     glitch_error
```

Fig. 4: Validation using instruction duplication

While this approach is simple to implement it is flawed, especially when using modern microcontrollers with multi stage pipelines. As shown by Yuce et al. in [8] injecting a single glitch can affect multiple instructions. This is possible, because the two load instructions are not executed one after another, but rather go simultaneously through various stages in the processor pipeline.

In general placing the validation of an instruction too close to the instruction itself renders the validation vulnerable to single glitch attacks.

B. Loop count validation

In [10], Proy et al. describe an automated compiler based glitch detection mechanism. Instead of validating arbitrary expressions as shown later in this paper, the approach from [10] focuses on validating loop exit conditions and iteration counts. The goal is to prevent attacks which weaken the security of cryptographic algorithms by reducing the number of encryption rounds.

A special compilation pass is added to LLVM, a very common compiler infrastructure. When encountering a loop with a iteration variable this optimization pass add a a second iteration variable which gets incremented or decremented the same as the original variable and thus allows to validate the loop exit condition after the loop exited. For example, the loop shown in 5a is modified to include a second variable and a condition check turning it into code for the loop shown in 5b.

```
int i = 0;
while(i < 10) {
  // ...
  i++;
}
```

(a) Loop with iteration variable

```
int i = 0;
int j = 0;
while(i < 10) {
  // ...
  i++;
  j++;
}

assert(j >= 10);
```

(b) Loop from 5a with validation

Fig. 5: Basic loop validation example

This optimization works best for loops with simple iteration calculation, i.e., adding or subtracting a constant from the iteration variable each iteration. Loops which contain `break` statements or which use a complex iteration modification however increase complexity of correct validations. The code listings in Figure 6 demonstrate these special loop forms.

A glitch attack on the calculation of `x` in Figure 6b would affect not only the iteration variable, but also a possible validation variable. Thus, for glitch robustness not only the iteration variable needs to be duplicated and recalculated, but also all variables used to modify it. In [10] this is achieved by tracing through the expressions used to modify the iteration variable and recalculating all these expressions.

The following Section describes a similar, but broader approach, which not only validates loop conditions but rather all expressions calculated in a function.

```
int i = 0;
while(i < 10) {
  // ...
  int x = // ...
  if(x == 42)
    break;
  i++;
}
```

(a)

```
int i = 10;
while(i > 0) {
  // ...
  int x = // ...
  i -= x;
}
```

(b)

Fig. 6: Advanced loop validation examples

C. Mathematical validation of computed values

The paper [5] by Aumüller et al. focuses on protecting the Rivest-Shamir-Adleman (RSA) cryptosystem. Instead of classical instruction duplication or expression validation techniques as described in Sections III-A and III-B it makes modifications and validations on a higher level. Rather than looking at algorithm implementations or even machine code, the computation is validated by modifying the RSA CRT algorithm, a common optimization to RSA using the Chinese Reminder Theorem (CRT).

Given a message m, two primes p and q, and the exponent d, the regular RSA CRT algorithm consists of the following steps to calculate the signature S of a message m:

$$d_p = d \mod (p-1) \tag{1}$$

$$d_q = d \mod (q-1) \tag{2}$$

$$S_p = m^{d_p} \mod p \tag{3}$$

$$S_q = m^{d_q} \mod q \tag{4}$$

$$S = S_q + q \cdot ((S_p - S_q) \cdot q^{-1} \mod p) \tag{5}$$

Aumüller et al. [5] extend the calculation of S_p and S_q shown in (3) and (4) with a third, smaller prime t and two random numbers r_1 and r_2:

$$p' = p \cdot t \tag{6}$$

$$d'_p = d_p + r_1 \cdot (p-1) \tag{7}$$

$$S'_p = m^{d'_p} \mod p' \tag{8}$$

$$q' = q \cdot t \tag{9}$$

$$d'_q = d_q + r_2 \cdot (q-1) \tag{10}$$

$$S'_q = m^{d'_q} \mod q' \tag{11}$$

The two values S'_p and S'_q can then be used to calculate S_p and S_q and thus S using Equation (5).

$$S_p = S'_p \mod p \tag{12}$$

$$S_q = S'_q \mod p \tag{13}$$

This modified algorithm allows validation of the result using the conditions shown in equations (14) to (19). [5]:

$$0 \equiv p' \mod p \tag{14}$$

$$0 \equiv q' \mod q \tag{15}$$

$$d_p \equiv d'_p \mod (p-1) \tag{16}$$

$$d_q \equiv d'_q \mod (q-1) \tag{17}$$

$$0 \equiv S - S'_p \mod p \tag{18}$$

$$0 \equiv S - S'_q \mod q \tag{19}$$

While this approach allows to implement a glitch hardened RSA implementation it only applies to RSA. It is not a general approach at hardening algorithms against glitch attacks. While its principles could be applied to other algorithms each requires manual work by a developer, rather than e.g., automatically applying protection using a compiler pass as shown in the following section.

IV. Detecting Glitches using Expression Validations

Traditionally, glitch detection techniques use instruction duplication or even triplication. While this works for some architectures, as described in Subsection III-A, a duplicate instruction is still vulnerable to a single fault on processors featuring a multi stage pipeline. Thus, in order to increase the robustness of glitching detection mechanism, the validation has to be placed as far away from the original computation as possible. In compiler engineering functions are divided into multiple blocks through which execution flows linearly. Moving validations out of the basic block of the original computation, means the number of instruction executed between computation and validation can vary between just a few computations to multiple calls to other functions. Placing validations farther away from their original computations makes it harder for an attacker to glitch both computation and validation.

The following sections, based on the short version of this paper [1], describe how to find the optimal locations for validations and how to validate both computations and conditional branches.

A. Identifying Locations for Validations

As described in Subsection III-A glitch detection mechanisms are still vulnerable to a single glitch fault when the duplicated instruction, in our case the second computation, is placed close to the original instruction. Placing the validation as far away from the original computation as possible ensures its robustness against single fault attacks.

The last possible location for a validation check is usually the end of the scope a value is defined in. For a value defined in a conditional or loop body this results in the check being placed at the end of the conditional or loop respectively. For a value defined in a function the last possible check is right before the function returns. Figure 7 shows an example with these two cases.

```
int main(int argc, char **argv)
{
  int x = argc * 10 - 2;
  if(argc > 1)
  {
    int y = x * 3;

    if(argc > 2)
      puts(argv[1]);

    // <-- validate `y` here
  }

  // <-- validate `x` here
  return x;
}
```

Fig. 7: Example Code

While it is trivial to find the optimal location for immutable variables in program code, a mutable variable might be changed between its first initialization and the end of the scope. In order to correctly validate all values of a mutable variable the location has to be determined during a later stage in the compilation process. The Static Single Assignment (SSA) form is a very common form of representing a program in compilers. In SSA form each variable is immutable and only assigned once, variables which are originally mutable and set multiple times are split up into seperate variables for each assignment. Additionally, a function in SSA form is usually represented as basic blocks rather than loops and branches. Figure 8 shows how *gcc* represents the code listed in Figure 7 internally after SSA creation.

The validation of `x`, labeled `x_5` in Figure 8, can be placed in block 5 (B_5). But there does not exist a block for the optimal location to validate `y_7`. It cannot be placed in B_5, as that block is also reachable from B_2 where `y_7` does not exist. Thus a new block has to be created, with B_3 and B_4 as predecessors and B_5 as successor. The edges $B_3 \rightarrow B_5$ and $B_4 \rightarrow B_5$ have to be removed. The validation of `y_7` can then be placed inside the newly created block.

In general, a variable x created in block B_x can only be validated in B_x itself or in a block B_i where all predecessors $prec(B_i)$ are direct or indirect successors of B_x. The optimal location for the validation is by definition the block that is the farthest away from B_x while still meeting the required condition.

Figure 9 shows the SSA block graph of Figure 7 with validations. Block B_5 is the newly inserted block and B_6 the former block 5.

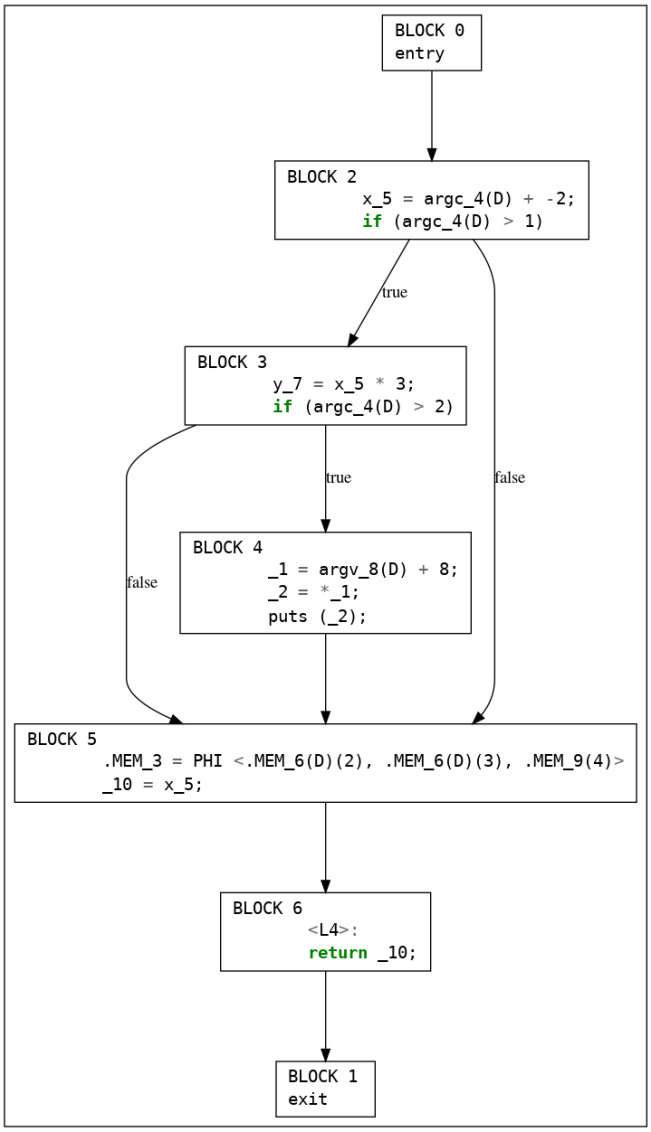


Fig. 8: Basic Block graph in SSA form of 7 without validations

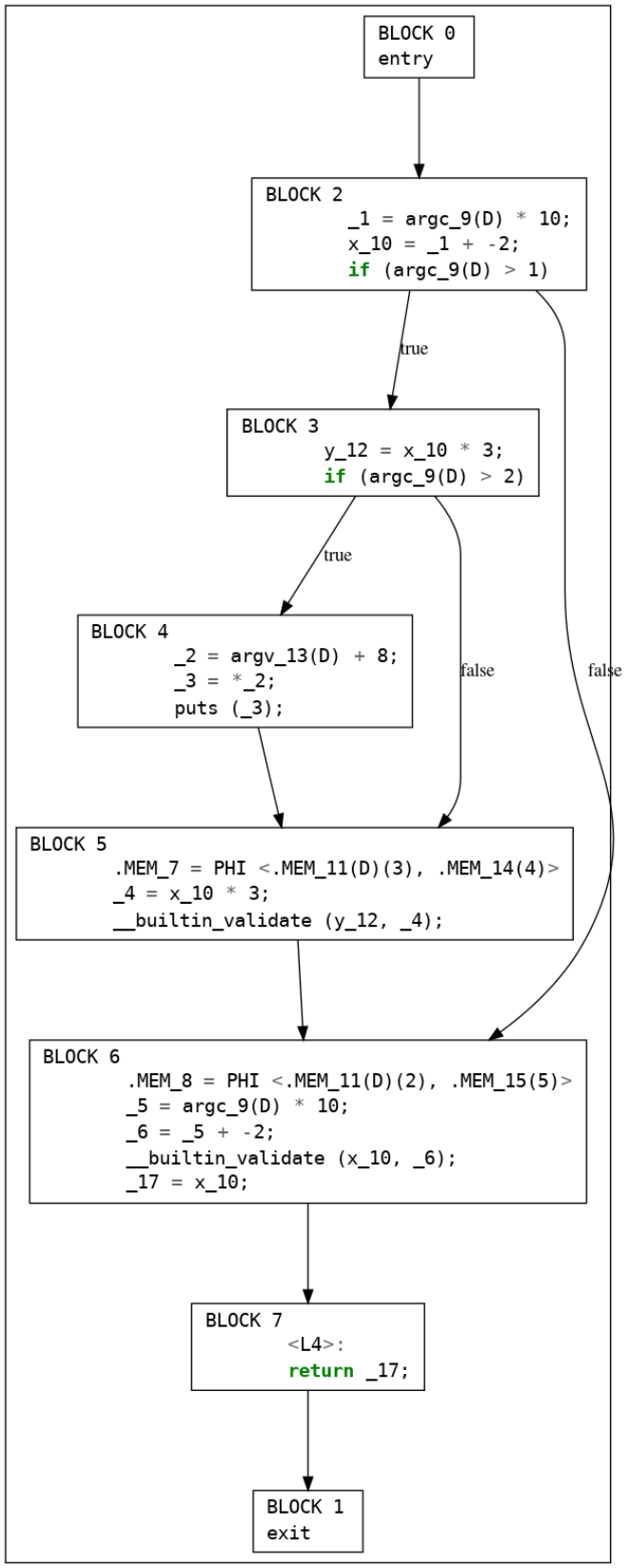


Fig. 9: Basic Block graph in SSA form of 7 with validations

B. Validating Calculations

Without deeper knowledge of the implemented algorithm validating calculations often boils down to simply recomputing all values and thus duplicating the entire calculation.
For example, the statement `int x = argc * 10 - 2;` from Figure 7, results in the SSA shown in the following listing:

```
_1 = argc_9(D) * 10;
x_10 = _1 + -2;
```

For a full validation both the SSA values `_1` and `x_10` have to be recalculated and validated:

```
_5 = argc_9(D) * 10;
__builtin_validate (_1, _5);
_6 = _5 + -2;
__builtin_validate (x_10, _6);
```

A simpler approach is to only validate the outermost result of one or more chained calculations. For the above example this is achieved simply by removing the first instance of `__builtin_validate` resulting in the code shown in 9. For larger entangled calculations removing redunant validations allows to greatly reduce the amount of validations required. For instance all variables in the following C code can be validated using a single validation of `z` instead of having to validate all variables or even all intermediate SSA values one by one.

```
int x = a * 10 + 3;
int y = x / 7;
int z = x * y * 13;
```

The `__builtin_validate` function acts similar to an assert equals function, it continues with execution if the two values are identical and cancels execution otherwise. In a production environment the function can be inlined producing an inequality check and a conditional jump to an error function, resulting in code similar to what gcc produces for calls to `assert`. Figure 10 shows the validation of `x` from Figure 7.

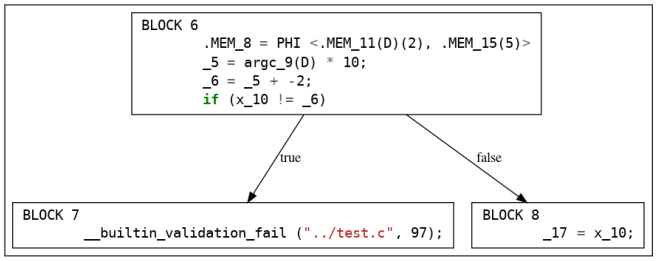


Fig. 10: Validation in production

C. Validating Comparisons and Conditional Jumps

In *gcc* the condition of a branch can not only be a single SSA value, but also a comparison operation. An example is the `if (argc_4(D) > 1)` statement at the end of block 2 in Figure 8. This is because in most processor architectures

a comparison of two values used for a conditional jump is done without storing the result in a common register, i.e., the comparison result is only stored in a flags register, which is then immediately used by the following conditional jump instruction.

As there exists no SSA name for the result of such comparisons in *gcc* it cannot be validated as described in Subsection IV-B. A block with a conditional branch at the end always has two successors, one for when the condition is true, one for when its false. Therefore, in order to validate the condition, two validations, one for each successor have to be created. Each validation follows the same rules as described in Subsection IV-A with their initial blocks being the targets of the conditional edges.

In general, for a block B_i with multiple successors, the branching condition can be validated using one validation placed as if a value j has been created in B_j for all edges $B_i \rightarrow B_j$.

If one of the successors B_j is also a direct or indirect successor of any of the other successors of B_i a new block between B_i and B_j has to be inserted. This is usually the case for loops and if statements without an else block. For example, in Figure 8 the validation for the condition of B_2 being false cannot be placed in B_5, as B_5 is also a successor of B_3.

D. Performance Considerations of Expression Validations

Simple instruction duplication mechanisms as described in III-A duplicate the runtime of the protected instructions. This holds true for simple microprocessors where each instruction takes a fixed amount of clock cycles. For advanced processors, which incorperate memory caching, a second load of a specific address will result in a cache hit, which is usually faster than a load from memory.

The novel glitch detection approach described in Section IV also duplicates instructions and thus has similar effect during runtime. The bigger impact, howver, is its prevention of possible compiler optimizations resulting in the generation of less performant instructions. Normally a compiler analyzes the lifetime of variables and the collisions between those lifetimes. The lifetime of a variable starts when the variable is first set and ends with its last usage. Two lifetimes collide when they are both alive at any given point in the function. When two lifetimes do not collide they can be placed in the same processor register. With too many lifetime collisions the compiler might run out of registers to assign and has to place variables in memory instead [7]. By definition, the optimal location for validation, as given in Subsection IV-A, extends the lifetime of variables to the maximum possible. Thus, with the novel detection approach, the register allocator of the compiler will have to place variables in memory more often, resulting in more memory accesses and decreased performance.

For example, the SSA variable `y_12` of Figure 9 would normally live only for a short time in B_3. Its validation in B_5 extends its lifetime, making it collide with the SSA variables `_2`, `_3` and `_4`.

In order to decrease the performance impact expression validation can only be enabled for security relevant functions such as cryptographic implementations or credential checks by disabling validations for all functions and adding a special compiler attribute to relevant ones.

V. Proof of Concept Implementation of Expression Validations

In order to test the feasability and actual performance impact of expression validations, the validations as described in Section IV were implemented into an existing compiler backend. The compilation library *GNU libjit* was picked as a framework for the implementation, as its internal immediate representation (IR) is simple and it already includes the liveness algorithms described in [7]. The liveness information will also be used in the following sections in order to implement the validation placement as described in Section IV-A.

A. libjit Immediate Representation

A compilation unit in libjit is a function. Each function consists of one or more basic blocks. Each basic block contains instructions, which are executed linearly. Having a call, jump or conditional jumps ends the current basic block and starts a new one. This way execution always flows linearly through a basic block. A block a, which may jump to another block b, is called bs *predecessor* and a is called a *successor* of b. Each block can have multiple *successors* as well as multiple *predecessors*. The *control flow graph (CFG)* of a function is a representation of the functions control flow using a directed graph G with a node v_i for each basic block i and edges v_{ij} for each successor v_j of v_i.

An instruction in libjits immediate representation consists of an operator `op` and up to three values. These values are named `dest`, `value1` and `value2`. Thus the usual form of an instruction is `dest = value1 op value2`. Complex mathematical expressions have to be broken down to a series of such instructions with the use of temporary values. For example the expression `y = 2 * x + 7` in libjit IR uses a temporary value `i0` as shown in the following listing:

```
i0 = 2 * x
y  = i0 + 7
```

There however exist many instructions where neither `dest` nor `value2` exist or just one of them. One example for such instructions are the conversion instructions, such as converting a 64-bit integer to a 64-bit floating point value denoted as `dest = long_to_float64(value1)`.

Each instruction includes flags what kind of operation is performed and which of the three values are used. For the purpose of expression validations only arithmetic and comparison instructions are relevant. These instructions can simply be cloned and given a new `dest` value, which is then compared to the original `dest`.

Libjit does not represent instructions in SSA form. This means a value used as a destination for an instruction might be used as a destination for other instructions too and thus its

value might change. This makes determining locations for expression validations harder, as a cloned instruction might depend on a value changed between the original computation and the duplicated one.

B. Liveness Flags in libjit

Libjit implements liveness and dataflow analysis as described by Cooper and Torczon in [7]. Each basic block is fitted with flags about *killed*, *upwards exposed* and *living out* values. A value k is *killed* in block v_i if its value is changed inside v_i. A value is *upwards exposed* when it is used without being or before being *killed*. A value k *lives out* a block v_i when one of v_i successors has k *upwards exposed*.
In libjit these three attributes are stored in bitsets for each block, with one bit for each value that exists in the function. As libjit expressions are not in SSA form, for expression validation mainly the *kill* set of a block is relevant, that is when trying to validate an instruction in the form of `c = a + b` the validation cannot be placed after a modification of `a`, `b` or `c`.

C. Basic Block Domination

In Graph theory a node v_i in control flow graph G is said to be dominated by another node v_j when all possible paths from the entry point to v_i go through v_d. Thus in program execution when v_i is executed the instructions in v_d have been executed at least once.
Block domination is easily described in a recursive manner: A block v_j is dominated by itself and the intersection of all dominators of its preceeding nodes v_i:

$$Dom(v_j) = \{v_j\} \cup (\cap_{v_i \in preds(v_j)} Dom(v_i)) \qquad (20)$$

This recursive definition can be implemented using a fix point algorithm as shown in Figure 11. This direct implementation has a complexity of $\mathcal{O}(n^2)$ with n being the number of blocks in the control flow graph. In 1979 Lengauer et Tarjan described a faster algorithm for finding block dominators with a runtime of $\mathcal{O}(m \cdot log(n))$ with m being the number of edges and n the number of nodes [2].
With m and n staying below 1000 in all test cases and because the latter algorithm comes with a higher implementation complexity, the direct implementation, shown in Figure 11, was chosen for libjit.

```
dominators[v0] = {};

foreach vi in G:
  dominators[vi] = {v0, v1, ..., vn};

while dominators changed:
  foreach vj in G:
    dominators[vj] = \
      {vj} ∪ (∩vi∈preds(vj) dominators[vi]);
```

Fig. 11: CFG Dominator Algorithm

D. Finding Expression Validation Locations in libjit

Section IV-A described the optimal validation location, based on program code with the concept of scopes. In libjit IR the concept of scopes does not exist directly, as it operates on a lower level than regular program code. In program code a scope ends when the execution flow of multiple possible branches join without all branches being created within the given scope. In the example code in Figure 7 the scope where variable `y` was created ends with the end of the outer `if` statement. However the scope where variable `x` was created does not end at that location, as both the `if` and `else` branch were created within the scope of `x`.
When mapping the scope lifetime to basic blocks a scope of variable `k` is exactly alive in all blocks, which are dominated by the block creating `k`. This means a variable `k` can be validated in all blocks v_i where the creating block v_k of `k` is dominating v_i: $v_k \in Dom(v_i)$.
Additionally, as described in Section V-B, in libjit a block v_j is not a possible validation candidate when k is killed by one of its direct or indirect predecessors v_i, without v_i being the block with the original instruction that is to be validated. While applying these rules to all blocks gives us the list of *possible* validation locations what we are actually looking for is the *last possible* location for a validation. A block v_i is the last possible location for a validation of a value created in v_k when not all of its successors v_j are possible validation candidates for `k`, i.e., when not all successors are dominated by v_k.
When an instruction cannot be validated by any other block than the creating block itself the validation has to be placed at the end of the block. When the destination value or one of the operands is killed later inside the same block the validation has to be placed inside of the block, before the kill occurs.

E. Placing Expression Validations in libjit

In order to validate an instruction, first the original computation has to be recomputed and afterwards its result compared to the original computation result. In case of inequality a branch to a specific expression validation failure label is performed. Because of this conditional branch placing a validation actually starts a new basic block at the validation location. Adding new blocks invalidates the computed dominating block information as well as the control flow graph itself. Therefore, all expression validation locations are first computed and collected in a list before actually placing them into the libjit IR code.
Validations of instructions can either occur at the start or at the end of a basic block. Placing them at the start is easier, as a new basic block can simply be prepended before the first instruction. Placing validations at the end of a block requires moving the last instruction to another new block.

VI. Performance Impact of Expression Validations

As described in Section IV-D the expected runtime increase is at least 100%. This section measures the actual performance impact of the reference expression validation implementation in libjit. As libjit is simply a library used for compiling

low level instructions to machine code a higher level language called *PointerScript* is used for implementing the test algorithms. *PointerScript* is a language with JavaScript like syntax, but direct access to C functions through its built-in foreign function interface. For the purpose of performance measurements several common algorithms such as array sorting, simple addition loops as well as special worst case scenarios were implemented and ran both with expression validations enabled and disabled.

A. Performance Test Setup and Programs

As libjit is a JIT compilation library the total program execution time normally also includes the compilation steps of the program code. For the purpose of this paper only the actual code runtime after compilation was measured. Even though for programs running multiple seconds compilation time only takes a small amount of the runtime, this way the additional compilation time caused by expression validation placement is ignored.

The POSIX gettimeofday function is called before and after calling the compiled program code and the difference is used for measuring the runtime of the programs. Each test program is run ten times and their runtimes are averaged. As runtime differs based on processor model and speed, for comparison not the actual runtime, but rather the runtime increase in percent is used as a metric.

The first test program is a simple loop calculating the sum of numbers between 1 and n. Its code is shown in Figure 12

The second test program partly listed in Figure 13 is a textbook implementation of the bubble sort sorting algorithm applied to an array of n random integers generated using the `rand` function.

The last code snippet shown in Figure 14 is an example crafted especially to visualize the performance overhead introduced by expression validations. It includes six variables all initialized with a random value and used in a tight loop with a high iteration count. Between the initialization of the variables and the actual loop is a call to an external function. Because of this function call all values created before the call have to be placed either in callee saved registers or in memory. Without expression validations there are exactly six values used after the call, which is identical to the amount of callee saved registers in the System V ABI on the x86-64 architecture. With expression validations enabled the register allocator has to spill some of the values, i.e., place them in memory instead. Figures 15 and 16 show this effect in the disassembly of Figure 14 with validations disabled and enabled respectively.

```
var sum = 0;
for(var i = 0; i < 10000000000; i++)
{
  sum += i;
}
```

Fig. 12: Non-Gauß Addition

```
for(var i = 0; i < len; i++)
{
  for(var j = 0; j < len - 1; j++)
  {
    if(parts[j] > parts[j + 1])
    {
      var tmp = parts[j + 1];
      parts[j + 1] = parts[j];
      parts[j] = tmp;
    }
  }
}
```

Fig. 13: Bubblesort

```
var x = rand();
var a = x + 1;
var b = x + 2;
var c = x + 3;
var d = x + 4;
var e = x + 5;

printf("");

for(var i = 0; i < 100000000; i++)
{
  a++; b++; c++; d++; e++;
}
```

Fig. 14: Register Allocation Spill Triggering Code

```
48 ff c3  inc %rbx
49 ff c4  inc %r12
49 ff c5  inc %r13
49 ff c6  inc %r14
49 ff c7  inc %r15
```

Fig. 15: Disassembly of the loop body from Figure 14 with validations disabled

```
4c 8b 4d f8     mov -0x8(%rbp),%r9
4d 8d 51 01     lea 0x1(%r9),%r10
49 8b c2        mov %r10,%rax
48 89 45 f8     mov %rax,-0x8(%rbp)
4c 8b 5d f0     mov -0x10(%rbp),%r11
4d 8d 73 01     lea 0x1(%r11),%r14
49 8b c6        mov %r14,%rax
48 89 45 f0     mov %rax,-0x10(%rbp)
```

Fig. 16: Disassembly of just two of the increments from Figure 14 with validations enabled

B. Performance Impact of Expression Validations

Figure 17 shows the relative runtimes of the three code samples. Each runtime is normalized to the runtime of the program with glitch detection disabled.
As expected the minimum runtime increase is around 100%, i.e., the runtime is doubled compared to running the same code without expression validations. This is especially true for the code samples listed in figures 12 and 13, where the only impact of expression validations is the added second computation of expressions within the loop.
For the code sample listed in Figure 14 however the expression validations do not only duplicate computations, but also have an impact on the register allocator and thus result in a significantly worse performance decrease. Running the program with expression validations enabled results in an approximatily 4.5 times as long execution time. The forth pair of measurements in Figure 17 describes the runtime increase of 14 with libjits graph coloring register allocator disabled. The execution time in this case is given relativly to the code running with the advanced allocator enabled. The overhead factor is reduced to around 3, which is mainly caused by the non-validated program also storing and loading values from memory and becoming nearly two times slower, while the runtime with expression validations only increases from 4.5 to 5.5 times slower than the runtime with advanced register allocation and without expression validations.

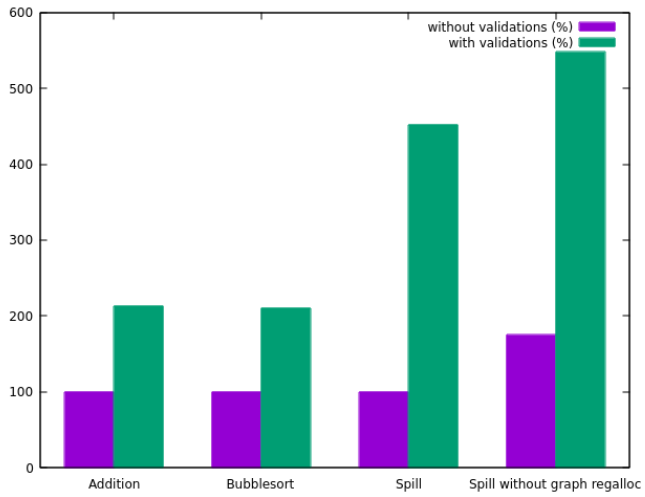


Fig. 17: Relative runtimes of 12, 13, 14 and 14 using only a basic register allocator

VII. Conclusion

After giving an introduction to glitching attacks and clock glitches in particular, we discussed various software based approaches at hardening against glitching attacks. While the common protection mechanism discussed in Subsection III-A can easily be applied to a program via an additional compilation pass, it is also shown to be ineffective [8]. The protection mechanism discussed in Subsection III-B by Proy et al. [10] can easily be applied to existing codebases, but only validates loop conditions and loop iterators.
The approach described in Section IV tries to combine the best traits of the three described previous mechanisms. It is similar to the mechanism by Proy et al. [10] as it also comes in the form of a compiler pass and it also adds validations of existing computations to the program. However, it not only validates loop conditions, but rather generalizes validation of arbitrary computations and branch conditions. This allows it to also protect the program from glitch attacks targeting value computations or substitutions, instead of only protecting against attacks aimed at modifying loop execution counts.
Section V discusses the steps of implementing the glitch detection technique, first described by Löw et al. in [1], into existing compiler architectures and gives details about the implementation into the GNU libjit compiler backend. Section VI shows the performance impact of this proof of concept implementation based on three representative code samples. It shows the assumptions about runtime usually doubles, as assumed in Section IV-D, but also shows the performance impact can be way worse in specific scenarios. Thus the approach is best applied only selectively to specific parts of a program, keeping performance impact low while still providing protection to curcial code parts.

References

[1] J. Löw, D. Bayerl, and H.J. Hof, ”Software Based Glitching Detection”, The Fifteenth International Conference on Emerging Security Information, Systems and Technologies (SECURWARE), Athen, pp. 41-46, 2022.
[2] T. Lengauer and R. E. Tarjan, ”A fast algorithm for finding dominators in a flowgraph”, ACM Transactions on Programming Languages and Systems Volume 1 Issue 1, pp. 121-141, 1979.
[3] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton, ”On the Importance of Checking Cryptographic Protocols for Faults”, Advances in Cryptology — EUROCRYPT ’97, pp. 37-51, 1997.
[4] D.I. Crecraft and S. Gergely ”Analog Electronics - Circuits, Systems and Signal Processing” Elsevier Science, 2002.
[5] C. Aumüller, P. Bier, W. Fischer, P. Hofreiter, and J.-P. Seifert, ”Fault Attacks on RSA with CRT: ConcreteResults and Practical Countermeasures”, Cryptographic Hardware and Embedded Systems - CHES 2002 pp. 260-275, 2002.
[6] J. Balasch, B. Gierlichs, and I. Verbauwhede, ”An In-depth and Black-box Characterization of the Effects of Clock Glitches on 8-bit MCUs”, 2011 Workshop on Fault Diagnosis and Tolerance in Cryptography, 2011, pp. 105-114, 2011.
[7] K. D. Cooper and L. Torczon, ”Engineering a Compiler”, 2nd edition, Elsevier Science, 2012.
[8] B. Yuce et al., ”Software Fault Resistance is Futile: Effective Single-Glitch Attacks”, 2016 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC), pp. 47-58, 2016.
[9] S. Patranabis, A. Chakraborty, and D. Mukhopadhyay, ”Fault Tolerant Infective Countermeasure for AES”, J Hardw Syst Secur 1, pp. 3-17, 2017.
[10] J. Proy, K. Heydemann, A. Berzati, and A. Cohen, ”Compiler-Assisted Loop Hardening Against Fault Attacks”, ACM Trans. Archit. Code Optim. 14, 4, pp. 1-25, 2017.
[11] B. Selmke, F. Hauschild, and J. Obermaier, ”Fault Injection into PLL-Based Systems via Clock Manipulation”, Proceedings of the 3rd ACM Workshop on Attacks and Solutions in Hardware Security Workshop, pp. 85-94, 2019.