

Design Document

Off the box SMU blast Calculator



April 19, 2021

Cisco Systems

SEZ Unit, Cessna Business Park, Kadubeesanahalli Village, Hobli, Sarjapur, Varthur Rd, Marathahalli, Bengaluru, Karnataka 560103

Table of Contents

[1. Product Description 2](#_Toc69721827)

[2. Introduction 2](#_Toc69721828)

[3. Detailed Description 3](#_Toc69721829)

[4. Software Design Considerations 3](#_Toc69721830)

[4.1 Code Sharing and Reuse Design 3](#_Toc69721831)

[4.2 Functional Structure 3](#_Toc69721832)

[4.2.1 Modules and brief overview: 3](#_Toc69721833)

[4.2.2 Details of major modules: 4](#_Toc69721834)

[4.3 System Flow 10](#_Toc69721835)

[4.3.1 Flow control diagram: 10](#_Toc69721836)

[4.3.2 Data Flow Diagram: 11](#_Toc69721837)

[4.4 Data Structure Design 12](#_Toc69721838)

[4.4.1 Reverse Requirement Graph (Directed Graph) 12](#_Toc69721839)

[4.5 Working 14](#_Toc69721840)

[4.5.1 Finding the ELFs and Their Dependencies: 14](#_Toc69721841)

[4.5.2 Building the Graph 14](#_Toc69721842)

[4.5.3 Finding affected programs 15](#_Toc69721843)

Out of Box SMU Blast Radius Calculator

# 1. Product Description

This product serves as a utility tool for Cisco customers for checking whether an RPM package or shared library of the IOS can be replaced and what will be the impact of such a change on the state of the device. It aims to generate statistical analysis of what programs are going to be affected and how it is going to impact the network as routing tables and protocols may need restarting.

# 2. Introduction

When a new version of software packages or shared libraries are installed on an IOS device, the packages are updated in the secondary storage but the system keeps using the older version of that package that has been loaded into memory previously. This introduces certain problems:

1. If the package contains some critical bug fixes the system may remain in an unstable state, prone to crashes, errors and failures, until the system is rebooted.
2. In case of the package containing certain security patches, if the newest version is not loaded the system may remain vulnerable to exploits.
3. If a new functionality is added and the network needs to take advantage of it at that moment, it cannot be done because the new libraries are not loaded into memory.

The solution is to restart all the processes that are currently using the packages but doing so introduces challenges of its own. For instance, the package under consideration is being used by a routing protocol. Changing the package and restarting the process may lead to network downtime and the routing tables may need to be re populated. Similar scenarios can be imagined if the device is a firewall, distribution switch, etc.

We cannot use the tool directly on the Router because resources are very costly on a mission critical device. Running the tool live may use up system resources which will hinder the performance of the device and affect the network as a whole. Hence, We are supposed to run the software on a different work station and take the .iso image and installable .rpm file and figure out what programs will be affected.

From a business perspective, a network downtime maybe out of question as it may incur huge losses.

Not only that, Cisco is known for the reliability and usability of its devices. The very purpose of using a Cisco device, from a business perspective, is to avoid losses due to device failure. At the same time, we cannot leave our devices prone to exploitation and crashes. Hence, restarting processes and taking down the network, albeit for a short period of time, is out of question.

So, the solution is to help Cisco customers take an informed decision on whether to upgrade the packages and if yes, provide them with information regarding the impact of a certain upgrade.

The user may wait for a scheduled downtime to make the upgrade or may do the upgrade right away if no critical functions are affected.

# 3. Detailed Description

When a new version of software package is released the user needs to decide whether to do the upgrade or to wait for a scheduled upgrade time, on the basis of what programs and services are going to be affected and how critical they are to the network. This tool aims to provide them with that information in a combined tabular form.

This software, on receiving a new .rpm file or shared library checks what are the libraries that need replacing. Then it goes ahead and finds out the processes that are going to be affected.

The program gets all the programs that may be affected by the change and through what set of dependencies it is being affected, i.e., the shared library dependency tree.

Also, if there is an executable on the given .rpm file the tool also checks for its dependencies as well as, most likely, it will be affected.

After collecting the data, the data is presented to the user in a tabular form containing description about programs, services that will be affected. Additionally, for each process we will print the dependencies with respect to the dependency tree of shared libraries.

# 4. Software Design Considerations

This software aims to serve as a tool for calculating the SMU blast radius. It calculates the impact of a SMU update on the router and it does so by finding out if the SMU is a reload SMU and if not, then what are the processes that will be restarted.

**Design Goals:**

* **Modular Design to facilitate code reuse and better structuring.**
* **Scalable Architecture – the tool, owing to its highly parallel workflow can handle high amount of load with minimal performance impact.**
* **Multiprocessing – this tool can leverage multiple cores to reduce the time taken.**
* **Standalone design with minimal dependency requirements.**
* **Supports multiple inputs including multiple SMUs, RPMs and any combination of both.**

## 4.1 Code Sharing and Reuse Design

This is a standalone tool that uses Linux cli tools like rpm, isoinfo, objdump, etc. Some of the modules like the find module are very neutral in design and can be used in other software. It is developed with **IOS-XR and ncs5500** platform in mind.

## 4.2 Functional Structure

### 4.2.1 Modules and brief overview:

1. **Dependency Checker:**

Responsible for checking if all the smus and rpms required by the target rpm is already installed or not.

1. **Execution manager:**

Calls find module to get dependencies of each ELF file, builds the directed graph. It separated the SMUs on the basis whether it belongs to XR or SYSDMIN and processes accordingly.

1. **Find Module:**

Responsible for gathering information on elf binaries in the .iso(executables and shared objects) including: name, md5, instances, card(rp, lc) and the modules required for their execution and also for getting the shared objects on the SMUs or rpms.

1. **Graph Module:**

Responsible for building a graph out of the information gathered by find module; running dfs for finding impacted processes.

1. **Input\_Handler Module:**

Checks the input and gets required information from the find modules for each SMU and RPM that is provided. It makes use of **extract\_rpm.sh** module.

1. **Main Module:**

This is where the execution starts and is the module that brings together other modules.

1. **Output Module:**

Responsible for formatting the output and building the output tree that depicts exactly how a program is being affected.

1. **mount Module:**

Prepares the root file system of the router by unpacking the golden iso and extracting the rpm modules that are required to be installed. It uses two helper scripts: **get\_all\_root\_fs.sh** and **extract\_iso.sh.**

1. **Utools Module:**

Keeps all the helper functions in one place.

**Shell Scripts:**

1. **extract\_iso.sh**

extracts the golden .iso using isoinfo.

1. **extract\_rpm.sh**

extracts cpio archive of rpms

1. **get\_all\_root\_fs.sh**

responsible for extracting all root file systems: XR, sysadmin, NBI

### 4.2.2 Details of major modules:

#### Dependency Checker(dependency\_checker.py):

**Class Vertex:**

This class contains the name of a rpm that is available and different versions of it and what rpm objects make it available. For example:

**bgp:**

versions available:

|  |  |
| --- | --- |
| Version: | Provided by: |
| 2.0.0.0 | ncs5500-bgp-2.0.0.0-r663.x86\_64.rpm |
| 2.0.0.1 | ncs5500-bgp-2.0.0.4-r663.CSCvw33584.x86\_64.rpm |
| 2.0.0.2 | ncs5500-bgp-2.0.0.4-r663.CSCvw33584.x86\_64.rpm |
| 2.0.0.3 | ncs5500-bgp-2.0.0.4-r663.CSCvw33584.x86\_64.rpm |
| 2.0.0.4 | ncs5500-bgp-2.0.0.4-r663.CSCvw33584.x86\_64.rpm |

**Functions:**

|  |  |  |
| --- | --- | --- |
| **Name** | **Input/Output** | **Description** |
| **get\_appropriate** | I – version number  O – rpm corresponding to that version | This function takes a version number and returns the rpm object built from the rpm that makes that particular version available. |
| **add\_version** | I – version number and rpm object  O – None | We can add a rpm object and to an existing version and an rpm to a new version. |

**Class Graph:**

This class contains Vertex objects and helper function that can be used to check whether all the required packages are present or if any dependency is missing.

Functions

|  |  |  |
| --- | --- | --- |
| **Name** | **Input/Output** | **Description** |
| from\_provides | I – List of rpm objects  O – Graph object | Builds and returns the graph object. |
| get\_deps | I – target rpms, visited, unmet  O – list of rpm paths | This function takes a rpm and finds out if its dependencies are met.  unmet is a set that is populated with all the dependencies that are not met. |

#### Find Module(find.py):

**Classes:**

**Elf:**

This class is responsible for holding information on an ELF file found in rpms.

Attributes:

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| Name | string | name of the ELF file. |
| Rpm | string | rpm to which it belongs. |
| Card | string | card name to which it belongs (rp, lc, etc.). |
| Instance | string | card instance in which it is found (FRETTA-RP, BIFROST-RP, etc.). |
| Path | string | absolute path where this ELF resides. |
| md5 | string | md5sum of this file. |
| has\_startup | Bool | marks if this ELF file has a startup file. |

Methods:

|  |  |  |
| --- | --- | --- |
| Name | Input/Output | Description |
| \_\_init\_\_ | I –  Name: name of the ELF file.  rpm: rpm to which it belongs.  card: card name to which it belongs (rp, lc, etc.).  instance: card instance in which it is found (FRETTA-RP, BIFROST-RP, etc.).  path: absolute path where this ELF resides.  md5: md5sum of this file. has\_startup: marks if this ELF file has a startup file.  O – Elf object | Constructor |
| from\_path | I –  path: path to ELF file.  md5: md5sum of ELF file. | Builds an ELF object when path to the ELF file and its MD5 is provided. |
| check\_startup | I –  path: path to ELF file.  O – bool – if startup file exit. | Given the path to an ELF, checks if a startup file exists corresponding to the ELF. |

**Deps:**

Holds an ELF object and its dependencies together.

**Attributes**:

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| Elf | Elf object | Elf object |
| Deps | List | list of shared objects needed for the ELF to work. |

**Methods:**

|  |  |  |
| --- | --- | --- |
| Name | Input/Output | Description |
| \_\_init\_\_ | I –  Elf – Elf object  Deps – list  O – Deps object | Costructor |
| from\_path | I –  path – path to Elf object  O – Deps object | Builds Deps object from path to an ELF. |
| get\_deps | I –  path – path to ELF object  O –  List of dependencies | Get dependencies of an ELF if its path is given. |

**Rpm:**

Holds information on RPMs that maybe installed or is in a SMU that is being checked.

**Attributes**

|  |  |  |
| --- | --- | --- |
| **Name** | **Type** | **Description** |
| Name | String | complete name of the RPM file from which this object is to be made. |
| Pkgtype | String | This parameter along with pkgpresence and packagetype are used for sorting the RPMs into the order in which they are to be considered while looking for an ELF. for example, SMUs take precedence over base packages. |
| pkgpresence | String | refer pkgtype |
| packagetype | String | refer pkgtype |
| Deps | List | list of dependency objects found in the RPM. |
| rpm\_path | String | absolute path where the rpm package is. |
| Provides | Dictionary | packges that are made available by the rpm.  key -> package name.  values -> package version |
| Requires | Dictionary | packages that are required by this rpm to work.  key -> package name  values -> package version |
| pkg\_name | String | name of the rpm. rpm -qp --qf '%{NAME} %{VERSION}' |
| pkg\_version | String | rpm version. rpm -qp --qf '%{NAME} %{VERSION}' |

**Methods:**

|  |  |  |
| --- | --- | --- |
| **Name** | **Input/Output** | **Description** |
| \_\_init\_\_ | I –  name, pkgtype, pkgpresence, packagetype, deps, rpm\_path, provides, requires, pkg\_name, pkg\_version  O – Rpm object | Constructor |
| rpm\_precedence | I –  rpm1, rpm2 – Rpm objects  O – integer | Given two rpms, determines which rpm takes precedence. Used for sorting. |
| from\_path | Args  args[0]: string location where rpm is etracted.  args[1]: string location of unextracted rpm file.  O - Rpm object. | Builds the Rpm object when path to Rpm is given. |
| get\_pkg\_info | I –  rpm\_path – path to rpm file | Gathers information from the rpm file in three custom tags: PKGTYPE, PACKAGEPRESENCE, PACKAGETYPE. |
| from\_rpm | I –  rpm\_path – path to rpm file | Builds Rpm object when path to rpm file is given. |
| is\_elf | I –  path – path to elf file | Checks if a file is an ELF binary or not. |
| get\_elfs | I –  path – path to rpm extraction location | Gets all the elfs in a rpm. |
| Parse | I –  rpm\_path – path to rpm file | Builds a dictionary of packages that are made available by this rpm. |
| get\_reqs | I –  rpm\_path – path to rpm file | Builds a dictionary of packages and corresponding version required by this rpm. |

**Functions**

|  |  |  |
| --- | --- | --- |
| **Name** | **Input/Output** | **Description** |
| get\_rpms | I -  root\_fs  O –  list of Rpm objects | Given a directory having rpm files, Rpm objects out of it puts them in a list. |

#### Graph Module(graph\_builder.py):

**Classes:**

**Instance\_vertex:**

Works as the vertex of directed graph.

**Attributes:**

|  |  |  |
| --- | --- | --- |
| **Name** | **Type** | **Description** |
| Name | String | Name of the ELF file for which vertex is created. |
| Instance | String | name of the card instance to which the ELF belongs(FRETTA-RP, etc.,) |
| Elf | Elf object | Object of Elf class in Find module |
| Edges | List | list of Instance\_Vertices which will be affected if current vertex is modified. |

**Methods**

|  |  |  |
| --- | --- | --- |
| **Name** | **Input/Output** | **Description** |
| \_\_init\_\_ | I – Elf object  O – Instance\_vertex object | Constructor |
| add\_edge | I – Instance\_vertex object | Adds a new Instance\_Vertex object to edges attribute. |

**Vertex:**

Holds Instance\_vertices with same name but coming from different instances.

**Attributes:**

|  |  |  |
| --- | --- | --- |
| **Name** | **Type** | **Description** |
| name | String | name of the Instance\_Vertices |
| instances | Dictionary | key -> instance name  values -> Instance\_Vertex object |

**Methods:**

|  |  |  |
| --- | --- | --- |
| **Name** | Input/Output | Description |
| \_\_init\_\_ | I – name of Elf files | Constructor |
| add\_iVertex | I – Instance\_vertex object | Adds a new vertex from a different instance. |
| get\_iVertex\_dfs | I – Elf object | Used during dfs innitiation to find Instance\_Vertex corresponding to a elf in the target SMU i.e., checks if vertex with same name and instance exists and if md5s match. |
| get\_iVertex | I – instance name | Returns Instance\_Vertex corresponding to a preticular instance(inst). |

**Graph:**

This is a directed graph if vertex A points to B then if A is modified then B will be affected. Each vertex is an Instance\_Vertex object.

Attributes:

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| card | String | Identifies the card to which this Graph belongs(lc, rp, sc, etc.) |
| vertices | Dictionary | dictionary of Vertex objects. |
| visited | Set | vertices that have been visited during depth first searches. |

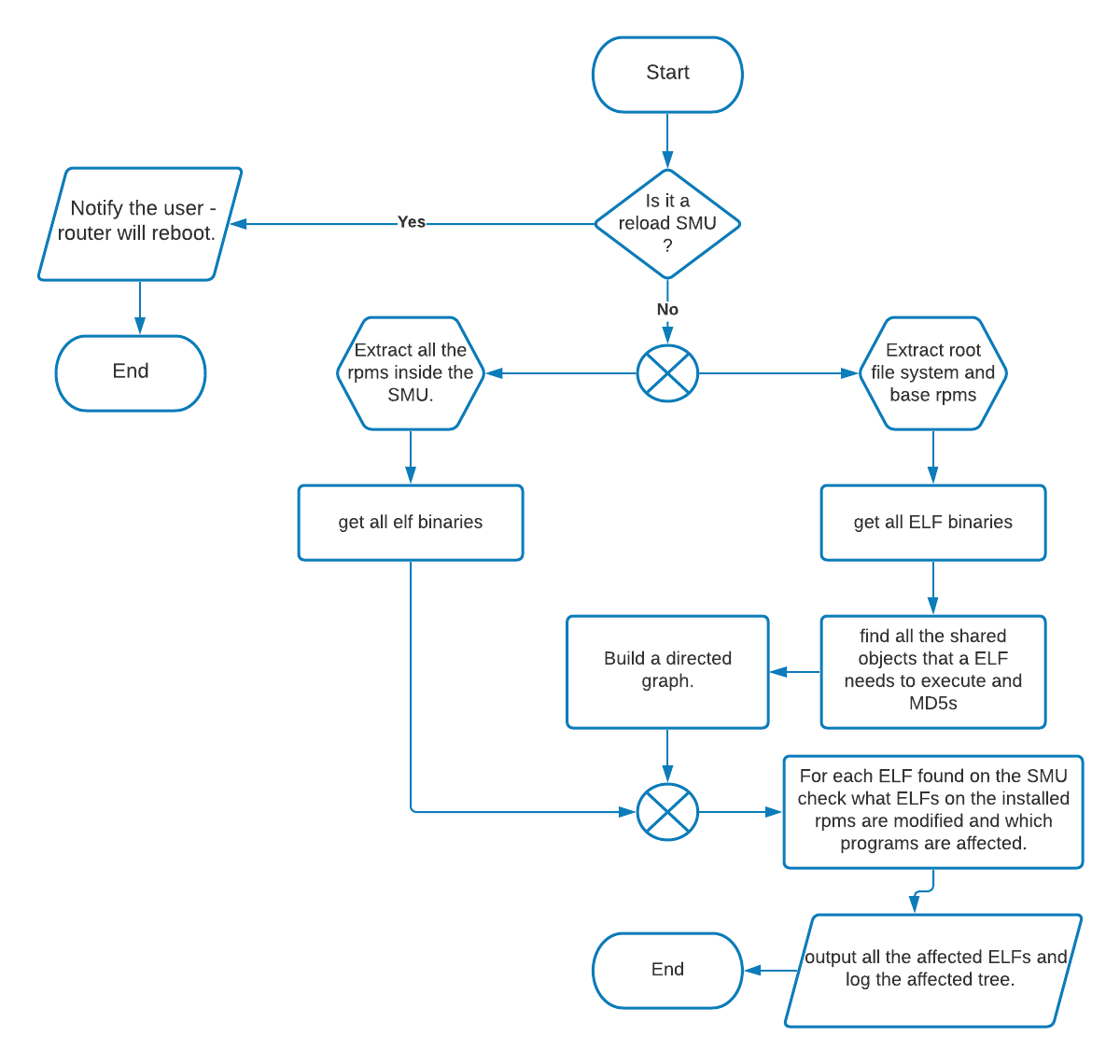
Methods:

|  |  |  |
| --- | --- | --- |
| Name | Input/Output | Description |
| \_\_init\_\_ | I – card name, Vertex object | Constructor |
| dfs | I – **iVertex** : Instance\_Vertex object  The target vertex for dfs.  **paths**: list  contains paths leading from a Instance\_Vertex that will be modified after SMU updateto each Instance\_Vertex found during dfs.  **affected**: list  constains all the vertices in the current dfs path. | Depth First Search on the graph to find all the affected vertices. |
| get\_affected | I – Elf object | Entry point for dfs. checks if Instance\_Vertex with same name and instance and different md5 exists; innitates dfs. |
| from\_deps | I – Dep\_list: list of Deps object  Card: card name | Builds a graph from list of Deps(dep\_list) objects for specified card. |

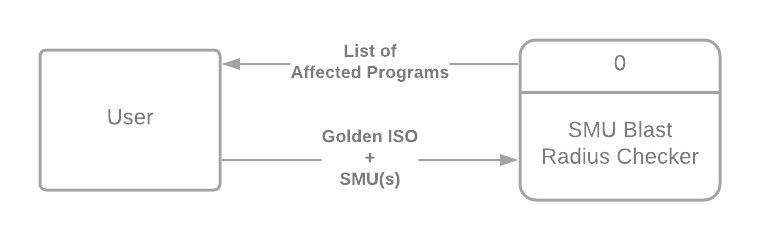
## 4.3 System Flow

### 4.3.1 Flow control diagram:

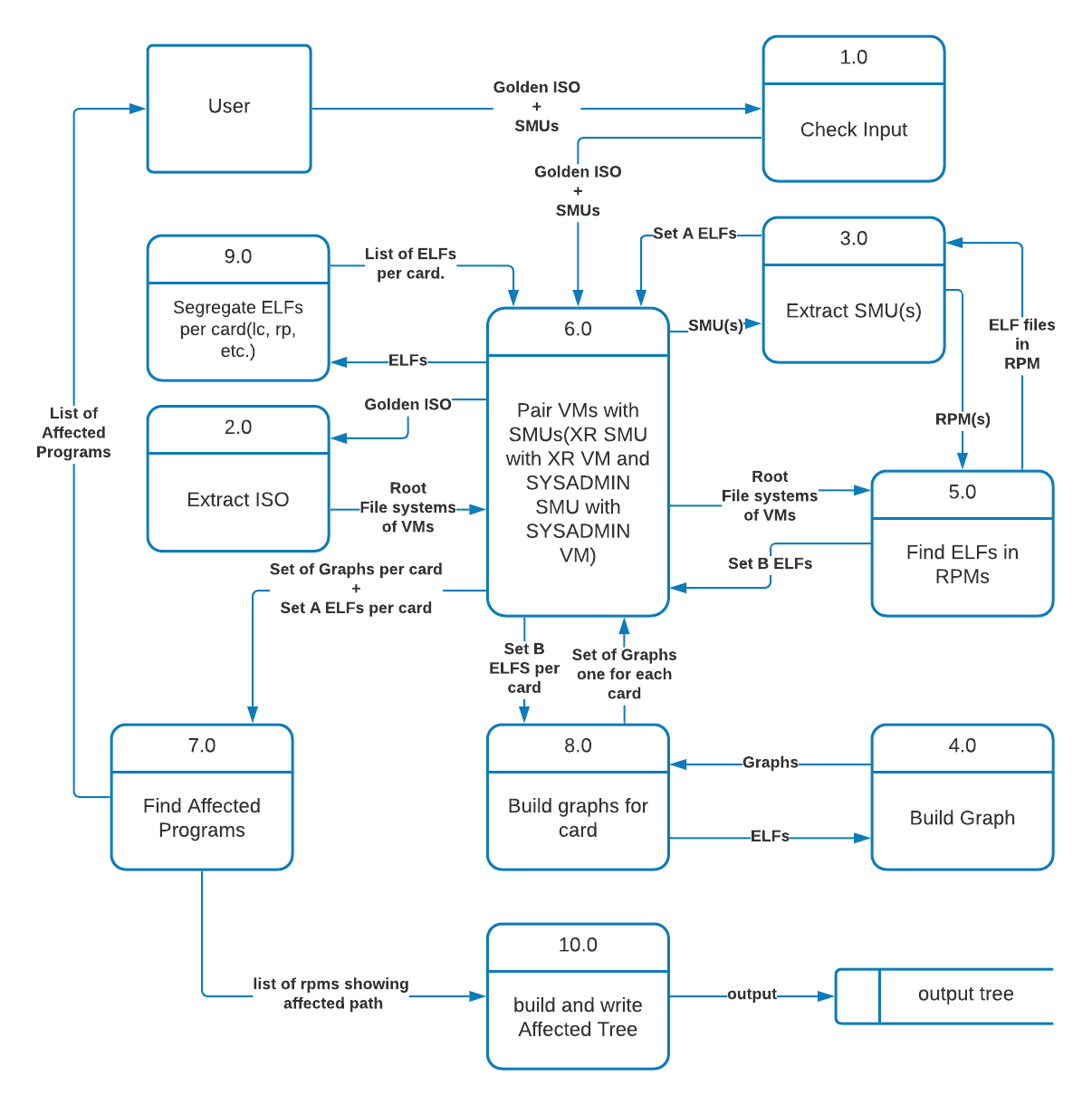
Overview of work process.



### 4.3.2 Data Flow Diagram:

Context Level Diagram

Level-0 Diagram:



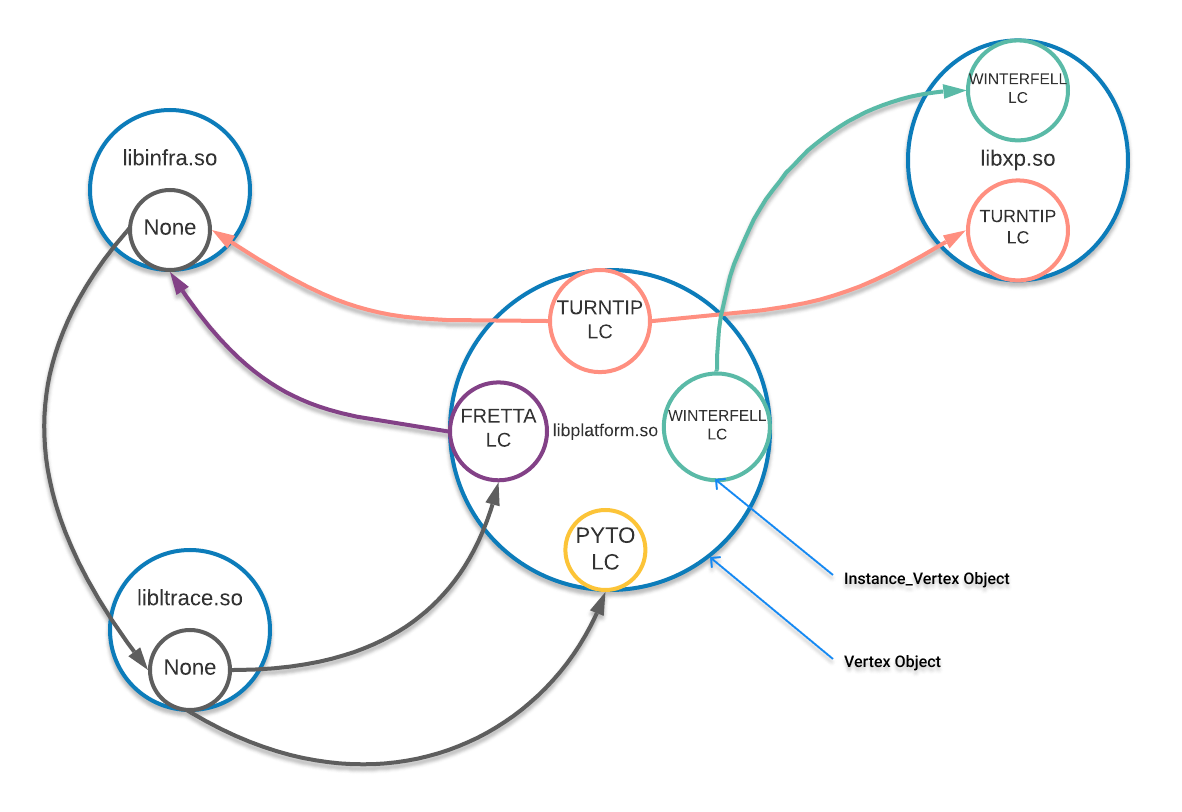
## 4.4 Data Structure Design

### 4.4.1 Reverse Requirement Graph (Directed Graph)

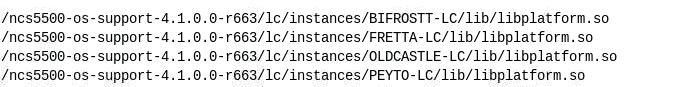
This is a directed Graph built on the logic that if ELF ‘A’ requires ELF ‘B’ to function then modifying B will force A to be reloaded and may initiate a chain reaction in which all the programs requiring A to function will also be reloaded/restarted.

Separate graphs are built for each card. For example, LC and RP will have their own graphs. These graphs are built in parallel to speed up the process.

Capable of handling multiple instances of same ELF file. For example, libplatform.so shared object may be available for multiple instances like FRETTA-LC, BIFROSTT-LC, etc., To handle this complexity the graph has multilayered structure as depicted in diagram below.

 Additional functionality: given an ELF file, program can find all the shared libraries needed for execution.

Every Instance\_Vertex object represent an ELF file. It has two things: an ELF file and a list of edges pointing to other Instance\_Vertex objects but multiple ELF files may have the same name. For example:



In order to accommodate same name ELF files coming from different instances, the Vertex object is used. We start building the graph by creating a new Vertex object each time we come across an ELF with a new name. We create an Instance\_Vertex for the ELF. Then we check if Vertex object with the same name exists or not. If it does then we check if there is already an Instance\_Vertex inside the Vertex with the same Instance and if it does then we do nothing but it doesn’t the we add the Instance\_Vertex under a new instance In the Vertex object.

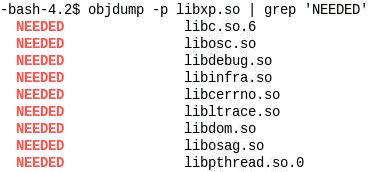
Now comes the part of building the edges. We go through the dependencies (shared objects) required by each ELF binary. For each dependency found, we probe the graph to check if a Vertex exists with the same name or not. If it does have, we check if there is an Instance\_Vertex of the same instance as the Instance\_Vertex. If its found then we add an edge pointing from the required Instance\_Vertex to the Instance\_Vertex whose dependences we are going through.

## 4.5 Working

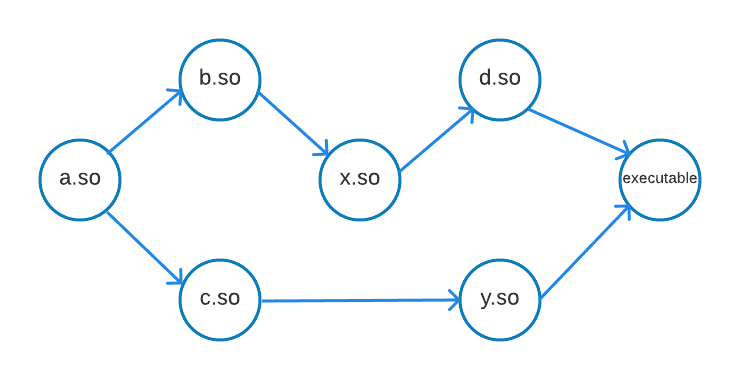
### 4.5.1 Finding the ELFs and Their Dependencies:

* First we extract the .iso to prepare the root filesystem of different VMs on the router. For example, we extract the ncs5500-sysadmin.iso; get the rpms inside it; extract the initrd.img inside it which inturn becomes the root file system.
* We go through each VM/LXC and in each rpm package that is installed we get all the files inside “/opt/cisco/XR/packages” or “/opt/cisco/calvados/packages” and then check if its an ELF binary by checking its magic number: “7f454c46”.
* There are multiple ways of finding dependencies of an ELF binary:

1. objdump
2. readelf
3. ldd
4. ELF processing libraries

* From this we chose objdump.
* The command ‘objdump -p [path/to/elf] | grep ‘NEEDED’’ is used to find the dependencies. For example:
* This complete process is done in parallel in order to improve performance.
* All the ELFs and its dependencies are bundled together and kept segregated on the basis of rpms.
* The rpms are sorted according to precedence. For Example, if an ELF is found in SMU and its Base package, then ELF in SMU takes precedence.

### 4.5.2 Building the Graph

* It’s a directed graph. The simplified diagram is shown below:
* For each VM/LXC we create multiple graphs, one for each card(lc, rp, etc.).
* In the above figure, A <- B –means changing B will affect A.
* This graph is capable of handling ELFs from multiple instances.
* The complete description of the graph can be found in Data Structure Design section.
* The following diagram shows how the dependencies of ELFs are used to build the graph:



### 4.5.3 Finding affected programs

* Depth First Search on the graph is used to find a path from a shared library that is changed by the target SMU to all the affected programs.
* The Depth First paths are stored to create an affected tree.

Depth First Search