Design Diversity for Private Network Model

Sumit Ahuja
Email: ahuja@alari.ch

Deepaknath Tandur K.
Email: tandur@alari.ch

Marina Sahakyan
Email: sahakya@alari.ch

Mehmet Guney
Email: guney@alari.ch

Abstract- Design Diversity has long been used to protect redundant systems against common mode failures. The general notion of diversity relies on independent generation of "different" implementations. In most systems, ensuring correctness is an important issue. In this project, we study the design of a communication network composed of secure and insecure channels and a central node that is responsible for interconnecting all branches belong to the network. We propose a design methodology that leads to correct design of this private communication network. This is accomplished by using the well-known technique of design diversity for building fault tolerant systems. Design Diversity takes two phases. First phase is called design phase, in which using different techniques for toleranting system design faults are explored. Second phase is called the comparison phase in which the outputs are compared and error is computed. Then the best design or selection of secondary design takes place. Especially, we will try to investigate different aspect of design diversity technique and then we will try to find out a suitable result to our application problem. It is shown for that most of the cases the technical challenge is to find out the difference of discrepancies for the output then using different techniques or some modification on these techniques to find out the appropriate solution. In the first stage we are looking at different techniques for design diversity. In the second stage we are applying these techniques to the chosen application and finding out an appropriate solution.

I. INTRODUCTION

Design diversity has been proposed in the past to protect redundant systems against common-mode failures [Avizienis 84][Lala 94] and has been used in both hardware and software systems [Riter95][Avizienis 77]. While most redundant sytems are designed using the single fault assumption, in real life there may be more than one sources of multiple and common mode errors which produce faults. Design diversity was described as a technique to avoid or tolerate common mode failures in redundant systems. In [Avizienis 84], *design diversity* was defined as the independent generation of two or more software or hardware elements (e.g., program modules, VLSI circuit masks, etc.) to satisfy a given requirement. The basic idea is that, with different implementations, common failure modes will produce different error effects. For example, chances of identical design errors may be minimized if two groups of designers are asked to independently design a hardware block or a software module. A dip in a power supply may have different effects on two different hardware implementations of the same logic function.

In this project, we study the design methodology of a private communication network. In this network, the users are connecting to each other using a common node, let's say a rooter in the between. Since each user uses this node to communicate with another user, there must be some rules for good communication. The users send requests to the middle node asking for connection to another specific user. When the first user asks for the establishment of a connection with a second user, the middle node checks the availability of the second user. In case it is not available the middle node decides to wait for a given time. When the availability status of the second user changes in this given waiting time then net step comes. Otherwise the middle node gives a time out and rejects the request. If the second user is available, then middle node checks if the request owner is belong to the defined group on the network. This rule is simply to avoid requests coming from a predefined group of members of that private network. For instance, some members of the network may not have access for that type of communication or their connectivity may be restricted for some reasons. The middle node has the list of these "banned" users or similarly has the list of users that are "not banned". In practice, the middle node checks if the address of the request holder is present in the list of users that are not banned. If the name is not present in this list, the middle node simply rejects the request. If the request owner has the necessary right for the connection then it is asked what kind of connection it requires. The type of connection may be secure or insecure. This is because there may be some secure channels in this private network so that some key-holders can connect each other over this secure channel. In case the request is for an insecure channel then the request is accepted and the connection is established. For the secure connection obviously secret keys should be used. However the required secret keys should be distributed to the respective users beforehand. Key distribution is out of

goal of this project and we simply assume that they are already distributed beforehand in some way. What our system does is to check the validity of the key and make a decision upon to establish the connection or to reject it according to the validity of the key.

The system we defined here is a typical example of the real communication networks used in today's life. Wireless private connections belong to this kind of systems. Because in private wireless connections there is a main base that connects users to each other using both secure and insecure channels. Radios may be thought of such a good example of this wireless networks. In a radio network one radio holder can connect to other holders over the main base and also can send encrypted messages using particular cryptographic algorithms. Moreover, the examples are not only restricted to radio networks. Such kind of applications that have these same characteristics is really high in number.

A. Why the correctness of this system is important?

Before going into details of the technique of design diversity, we want to emphasize the necessity for correctness of private networks. These kind of private networks are safety-critical systems since they are used in many businesses organizations and systems. Any error in this application may cause the break down of the connection leading to many losses. Another unwanted result can the leak of secret information. In addition this system must be resistant to attacks from outside world and provide security inside. Not only attacks but also some interference in the wireless network makes the reliability of such systems important. That's why in our case the middle node (or base station) asks the identity of the request owner and checks its relation with the network. The design of such network becomes much more important when the network size increases or secure communication becomes crucial.

In this example of network all incoming and outgoing packets between the users are passing through the main node. In other words the main node holds the burden of the communication problem and should be fast for obvious reasons. After checking the values of requests it should accept the true requests while rejecting the wrong ones. But the question is how to satisfy such correctness? The answer is, we must be so careful while defining the configuration of the network. However the configuration of the network is defined using mathematical terms and relations. How can we obtain a correct behavior from these mathematical rules in real world applications? Moreover the mathematical rules and relations that define the configuration are not unique. One solution to this design problem may be trial and error method. However this solution is not useful since it does not assure the correctness of the design. Because of the reasons stated above the cost of such trial and error method may be high in terms of both money and security. Assuring the correctness of a network configuration is therefore an important problem.

B. Design Methodology

According to Yang, Liu and Gouda [1] the diverse design methodology has two main phases: a design space and a comparison phase. In the design phase, the same requirement specification of the same problem is given to multiple teams who proceed independently to design different versions of the same network. In the comparison phase, the resulting multiple versions are compared with each other to find out the discrepancies between them. Then each discrepancy is further investigated and a correction is applied if necessary. The challenge in this diverse design methodology is how to discover all the discrepancies between them. They develop three algorithms for solving this problem: a construction algorithm for constructing an equivalent ordered decision diagram for a sequence of rules, a shaping algorithm for transforming two ordered decision diagrams to be semi-isomorphic without changing their semantics, and a comparison algorithm for detecting all the discrepancies between two semi-isomorphic decision diagrams.

Coming to the first phase of the problem, which is the design phase, we can say that different teams will approach to the problem in different ways. Because their experience, strentgh and background can not be same the design methods they proposed will be different.

Design diversity has been proposed in the fault tolerance literature to increase the reliability of the system. Design diveristy is defined as the independent generation of two or more different hardware of software elements to satisfy a given requirement [Avizienis84]. The main objective of design diversity is to protect redundant system from common-mode failures, which are failures that affect more than one module at the same time [Lala94]. Design diversity also has been applied to software systems. N version programming is one example of diversity in software. Design diversity in N version programming targets software design faults. In N version programming, different designers develop independent versions of the programs to avoid common design errors. By carefully chosing the teams that are assigned to solve the same problem we can provide that ne coincident errors appear in all versions. In the comparison phase, the resulting designs are compared and the discrepancies are found. The difficult step in this problem is how to discover all the discrepancies between the the rules or diagrams that define the system. According to Yang, Liu and Gouda [1], the solution for comparing two given set of rules or diagrams consists of the following three steps: (1) If either of the networks is designed by a sequence of rules, we construct an equivalent ordered network ordered network decision diagram from the sequence of rules by the construction algorithm shown in the following sections. If either of the two network is designed by a non-ordered network decision diagram, we

at first generate an equivalent sequence of rules from the diagram, and then construct an equivalent ordered network decision diagram from the sequence of rules. After this step, we get two ordered network decision diagrams. (2) If the two ordered network decision diagrams are not semi-isomorphic, we transform them to two semi-isomorphic network decision diagrams without changing their semantics by the shaping algorithm explained in the next sections. After this step, we get two semi-isomorphic network decision diagrams. (3) All the discrepancies between the two semi-isomorphic network decision diagrams can be discovered by the comparison algorithm.

In conclusion, in this project we do the following:

We make a taxonomy of the design diversity. Uses of design diversity techniques and multiple techniques are investigated.

We propose the desing diversity to ensure the correctness of a private network.

We search for the discrepancies between two given network definitions. The solution is found by using the so called algorithms: the construction algorithm, the shaping algorithm, and the comparison algorithm. In the end we are able to find a discrepancy so that these three algorithms are working efficiently.

II. TAXONOMY OF DESIGN DIVERSITY

Design diversity is a fundamental approach to the tolerance of design faults. It is applicable to all elements of an information system: hardware, software, and communication links, man/machine interfaces, design tools, etc. Design diversity is implemented by performing a function in two, three, or more independently designed elements (channels) and then executing a decision algorithm (a comparison or a majority vote) on the results. Frequently the comparison or vote has to be *inexact* because different algorithms are used by the diverse elements. A well protected implementation of the decision algorithms is the key requirement for successful application of design diversity. Since design diversity is a relatively costly technique, only the mission-critical parts should employ it in large and complex systems that are subject to hardware and software design faults. Examples of *software* diversity techniques are N-version programming and recovery blocks.

Software redundancy techniques can be divided into two major classes: (1) with diversity (2) without diversity. Diversity technique's aim is to tolerate design faults and data or design diversity. Design diversity is used to tolerate design faults in hardware and software. There are methods for tolerating software design faults:

- N-version Programming
- Recovery Blocks
- Back to back testing

1-) N-Version Programming

In this method, program versions are developed by N different programmers of different skills and objective is that N different programs fail independently. There is one voter which decides the program to be used for testing. This method uses voting on results produced by N program versions which are developed by different teams of programmers. The assumption here is that the programs fail independently.

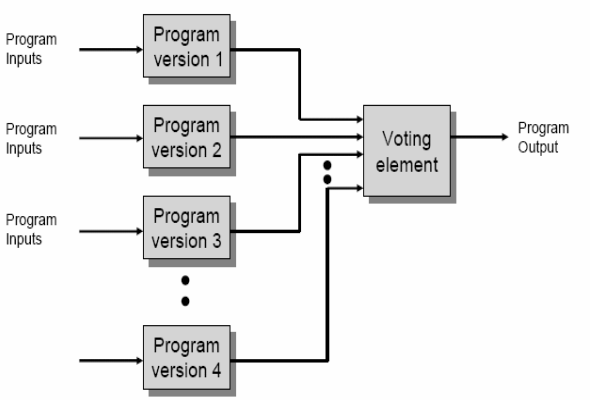

Fig.1: Block diagram for N-Version programming

2-)Recovery Blocks

Instead of using N different program versions here only one version which is called Primary version is used. It assumes that acceptance test will detect the errors. In case error occurs then secondary programs are used so that system should retain its fault tolerant property.

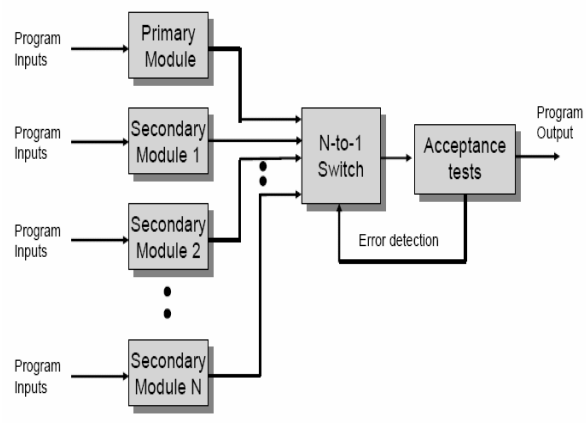

Fig.2: Block Diagram of Recovery Blocks.

3-) Back to Back Testing

Back-to-back testing is a complementary method to N-version programming. This method is used to test the resulting N versions before deploying them in parallel. The basic idea is as follows. At first, create a suite of test

cases. Then for each test case, do the following three steps: (1) Execute the Nprograms in parallel. (2) Cross-compare the N results. (3) Investigate each discrepancy discovered, and apply corrections if necessary.

III. DESIGN METHODS

In this project, we consider three methods which are explained in the section below. Before that we want to point out that the requests contains information of the address (e.g. IP address) of the request owner, the requested user, the type of connection (i.e. with or without encryption) and the secret key in case secure communication is chosen. The networks simply evaluates this information and makes a decision according to the given set of rules. As stated above, there are three main design methods: (1) Rule Based Design, (2) Recovery Blocks, (3) Back to Back Testing.

A. Rule Based Network Design

By this method, a network is defined by a set of rules. Each rule is a pair presented as predicate → decision. The predicate of a rule is a Boolean expression of the form $(F_1 \in S_1) \cap (F_2 \in S_2) \cap \ldots (F_d \in S_d)$ where $S_i \subset \Delta(F_i$ If .$)S_i = \Delta$ (F_i ecalper nac ew, $)(F_i \in S_i)$ by ($F_i \in$ all), or remove the conjunct ($F_i \in \Delta(F_i)$). Here is an example of a network rule: (N ∈ {Request}) ∩ (S ∈ all) ∩ (D ∈ (1)) ∩ (C ∈ (1)) ∩ (K ∈ (1)). Here S stands for the source address range, D for the destination user, C for the type of the communication request (i.e. secure or insecure communication). K stands for key information and N for an old or new request. To be more precise, the rule here says that if "the request is a new request" and "source address range belongs to the network" and 'destination user is idle or busy" and "the communication request is secure" and "secret key provided by the request owner is correct" then the connection is established by the network provider.

B. Diagram Based Network Design

Defining networks by decision diagrams is proposed by Gouda and Liu[3]. Here we use the idea of Network Decision Diagrams (NDD) over requests $F_1, F_2, \ldots F_d$. A NDD (Network Decision Diagram) is a rooted, directed and acyclic graph. An NDD has exactly one node that has no incoming edges and it is called the root of the NDD.

The nodes that have no outgoing edges are called terminal nodes, and the other nodes are called nonterminal nodes. Each node v is labelled with F(v), where we have F(v) ∈ {F1,F2,…Fd} if v is a nonterminal node, and F(v) ∈ {a,d} if v is a terminal node. A path from the root to a terminal node is called a decision path. No two nodes on a decision path have the same label. Each edge e is labelled with a nonempty integer set I(e), where I(e) ⊂ (F(e.s)), e.s is the source e and e.t is the target of e. The set of all the outgoing edges of a node v is denoted by E(v), which has the following two properties: (1) *consistency*: The integer sets labelled on any two edges e and e' in E(v) are disjoint: I(e) ∩ I(e') = Ø. (2) *Completeness*: The union of the integer sets of all the edges in E(v) is the domain of F(v): $U_{e \in E(v)} I(e) = \Delta(F(v))$.

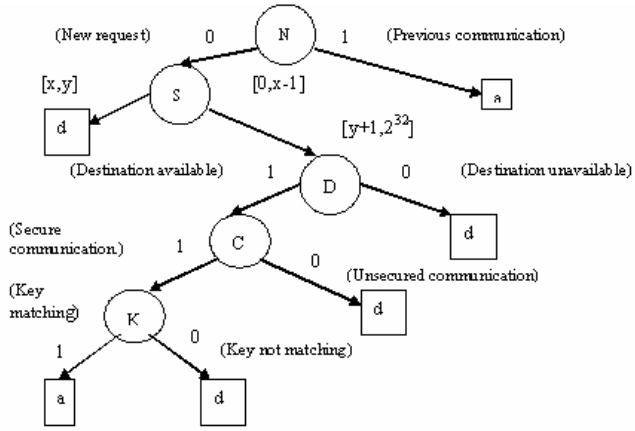

Fig3: Diagram based method

A decision path (v1 e1 v2 e2 …vk ek vk+1) of an NDD defines the following network rule: (F(v1) ∈ I(e1)) ∩ (F(v2) ∈ I(e2)) ∩ …∩ (F(vk) ∈ I(ek)) → F(vk+1). For an NDD f, we use Sf to represent the set of all the rules defined by all the decision paths of f. Because of the consistency and completeness properties of an FDD, for any packet, there is one and only one matching rule in Sf . For any packet p, an FDD f maps it to the decision of the unique rule in Sf that p matches. Given an FDD f, any sequence consists of all the rules in Sf is equivalent to f. The reason that the order of the rules does not matter is that there are no overlapping rules in Sf.

C. Flowchart Based Method

This method is very similar to the method of making flowchart in basic design. The flowchart shows the statements and conditions that are followed during the decision steps. Actually, the rules are hidden in the flowchart decisions. When a request comes, the decision is made according to the results of each block. By following the path according to the results of the conditions inside each block a decision is made.

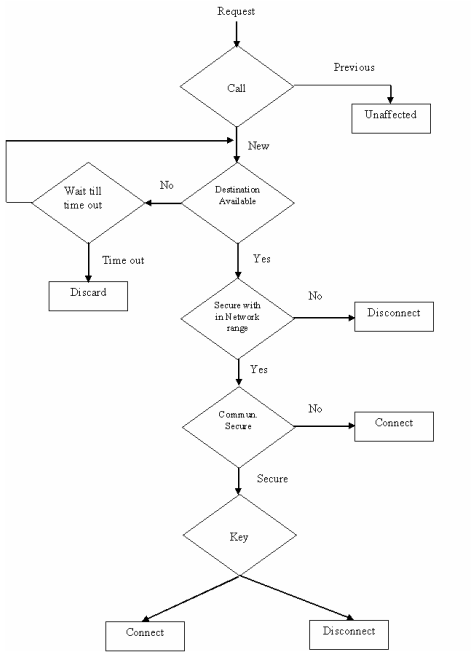

Fig4: Flowchart based method.

As we see, a configuration of a network can be defined in many different ways using different methods and rules, although they all try to implement the same design problem.

IV. CONSTRUCTION METHODS

In this section, we discuss how to construct an equivalent FDD from a sequence of rules <$r_1, r_2, \ldots, r_n$>, where each rule is of the format $(F_1 \in S_1) \cap (F_2 \in S_2) \cap \ldots \cap (F_d \in S_d) \rightarrow$ decision. In other words, all the d packet fields appear in the predicate of each rules, and they appear in the same order. We at first construct a partial FDD from the first rule. A partial FDD is a diagram that has all the properties of an FDD except the completeness property. The partial FDD constructed from a single rule contains only the decision path that defines the rule. Suppose from the first i rules we have constructed a partial FDD, whose root is v (v is labelled F_1, and v has k outgoing edges $e_1, e_2, \ldots, e_k$). Let r_{i+1} be $(F_1 \in S_1) \cap (F_2 \in S_2) \cap \ldots \cap (F_d \cap \in S_d) \rightarrow$ decision. Next we consider how to add rule r_{i+1} to the partial NDD.

At first, we examine whether we need to add another outgoing edge to v. If $S_1 - (I(e_1) \cup I(e_2) \cup \ldots \cup I(e_k)) \neq \emptyset$, we need to add a new outgoing edge with label $S_1 - (I(e_1) \cup I(e_2) \cup \ldots \cup I(e_k))$ to v because any packet whose F_1 field satisfies $S_1 - (I(e_1) \cup I(e_2) \cup \ldots \cup I(e_k))$ doesn't match any of the first i rules, but match r_{i+1} if the packet satisfies $(F_2 \in S_2) \cap (F_3 \in S_3) \cap \ldots \cap (F_d \in S_d)$. This new edge points to the root of the partial NDD built from $(F_2 \in S_2) \cap (F_3 \in S_3) \cap \ldots \cap (F_d \in S_d) \rightarrow$ decision.

Construction_Algorithm(f)

Input : *A firewall f of a sequence of rules $(r_1, r_2, \cdots, r_n)$*

Output: *An FDD f' such that f and f' are equivalent*

1. *Build a decision path with root v from rule r_1;*
2. **for** $i := 2$ **to** n **do** **Append**(v, r_i);

Append(v, $(F_i \in S_i) \wedge (F_{i+1} \in S_{i+1}) \wedge \cdots \wedge (F_d \in S_d) \rightarrow$ decision)

Input : (1) *Root v of a partial FDD. Node v is labelled F_p and it has k outgoing edges: $e_1, e_2, \cdots, e_k$*

(2) *Rule $(F_i \in S_i) \wedge (F_{i+1} \in S_{i+1}) \wedge \cdots \wedge (F_d \in S_d) \rightarrow$ decision*

Output: *The partial FDD with the above rule added*

1. **if** $((S_i - (I(e_1) \cup I(e_2) \cup \cdots \cup I(e_k))) \neq \emptyset)$ **then**

(a) *Add an outgoing edge e_{k+1} with label $S_i - (I(e_1) \cup I(e_2) \cup \cdots \cup I(e_k))$ to v;*

(b) *Build a decision path from rule $(F_{i+1} \in S_{i+1}) \wedge \cdots \wedge (F_d \in S_d) \rightarrow$ decision,*

and let e_{k+1} point to its root;

2. **if** $i < d$ **then**

for $j := 1$ **to** k **do**

if $I(e_j) \subseteq S_i$ **then** **Append**($e_j.t$, $(F_{i+1} \in S_{i+1}) \wedge \cdots \wedge (F_d \in S_d) \rightarrow$ decision);

else if $I(e_j) \cap S_i \neq \emptyset$ **then**

(a) $I(e_j) := I(e_j) - S_i$;

(b) *Add one outgoing edge e to v, and label it with $I(e_j) \cap S_i$;*

(c) *Replicate the graph rooted at $e_j.t$ and let e points to the replicated graph;*

(d) **Append**(e, $(F_{i+1} \in S_{i+1}) \wedge \cdots \wedge (F_d \in S_d) \rightarrow$ decision);

Second, we compare S_1 and $I(e_j)$ for each j in the following three cases:

1) $S_1 \cap I(e_j) = \emptyset$ In this case, we skip edge e_j because any packet whose value of field F_1 is in set $I(e_j)$ doesn't match r_{i+1}.

2) $S_1 \cap I(e_j) = I(e_j)$: In this case, for a packet whose value of field F_0 is in set $I(e_j)$, it may match one of the first i rules, and it also may match rule r_{i+1}. So we add $(F_2 \in S_2) \cap (F_3 \in S_3) \cap \ldots \cap (F_d \in S_d) \rightarrow$ decision to the subgraph rooted at $e_j.t$ in a similar fashion.

3) $S_1 \cap I(e_j) \neq I(e_j)$ and $S_1 \cap I(e_j) \neq \emptyset$: In this case, we split edge e into two edges: e' with label $I(e_j) - S_1$ and e'' with label $I(e_j) \cap S_1$. Then we make two copies of the subgraph rooted at $e_j.t$, and let e' and e'' point to one copy each. Thus we can deal with e' by the first case, and e'' by the second case.

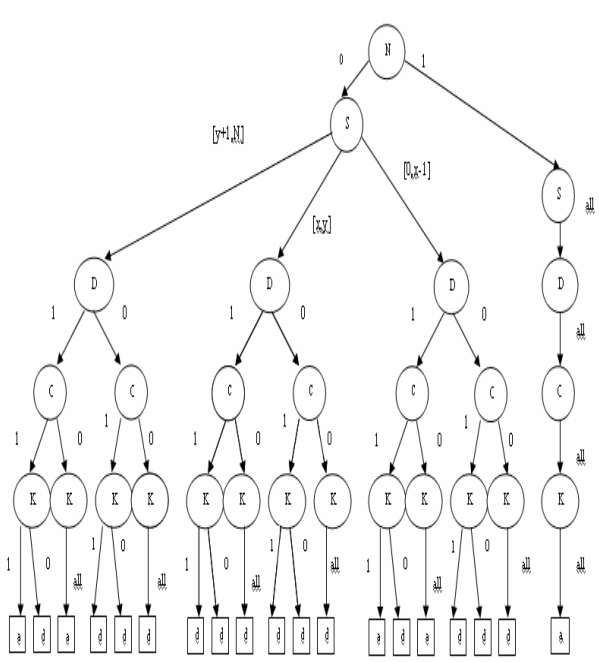

Fig5: Figure 3 after applying construction algorithm.

V. SHAPING ALGORITHM

In this section we discuss how to transform two ordered, but not semi-isomorphic NDDs f_a and f_b to two semi-isomorphic NDDs $f_{a'}$ and $f_{b'}$ such that f_a is equivalent to $f_{a'}$, and f_b is equivalent to $f_{b'}$.

Definition 5.1 (Ordered NDDs): Let $<$ be the total order over packet fields $F_1, F_2, \ldots, F_d$ where $F_1 < F_2 < \ldots < F_d$ holds. An NDD is ordered iff for each decision path $(v_1\ e_1\ v_2\ e_2\ \ldots v_k\ e_k\ v_{k+1})$, we have $F(v_1) < F(v_2) < \ldots < F(v_k)$. We also define $F_i <$ accept and $F_i <$ discard for $1 \le i \le d$.

By this definition, the NDDs constructed by the construction algorithm are ordered NDDs. For example, the NDD in figure 5 is an ordered NDD assuming F_1 is N (New request), F_2 is S (Source IP), F_3 is D (Destination IP), F_4 is C (request for secure communication), and F_5 is K (Secret Key). Therefore, if a given network is designed by a non-ordered FDD f, we can construct an equivalent ordered FDD f' from a sequence consisting of all the rules in S_f, where S_f is the set of all the rules defined by the decision paths of f. Since the sequence consisting of all the rules in S_f is equivalent to both f and f', f and f' are equivalent. Henceforth we can use f', instead of f, to compare with other networks.

Informally, two NDDs are semi-isomorphic if their structures are isomorphic, the labels of their corresponding nonterminal nodes match, and the labels of their corresponding edges match. In other words, only the labels of their terminal nodes may not match. Formally:

Definition 5.2 (Semi-isomorphic FDDs): Two FDDs f and f' are semi-isomorphic iff there exists a one-to-one function σ from the nodes of f onto the nodes of f', such that the following two conditions hold:

1) For any node v in f, either both v and $\sigma(v)$ are nonterminal nodes with the same label, or both v and $\sigma(v)$ are terminal nodes;

2) For each edge e in f, suppose e is from node v_1 and node v_2, there is an edge from $\sigma(v_1)$ to $\sigma(v_2)$ in f', and the two edges have the same label.

There are six types of transformations that can be applied to an NDD without changing its semantics. We call them NDD Equivalence Transformations.

1) *Node Deletion*: If a node v has only one outgoing edge e whose label is the domain of $F(v)$, then we can remove v and let all its incoming edges point to the target of e.

2) *Node Insertion*: If along all the decision paths containing node v, there are no nodes that are labeled with F, $F \in \{F_1, F_2, \ldots, F_d\}$, then we can insert a node v' with label F above v as follows: let all incoming edges of v point to v', create one edge from v' to v, and the edge is labeled the domain of F.

3) *Edge Merging*: If two outgoing edges e_1 and e_2 of a node v point to two isomorphic subgraphs, then we can merge these two edges into one edge as follows: delete e_2 along with the subgraph it points to, then change the label of e_1 to $I(e_1) \cup I(e_2)$.

4) *Edge Splitting*: For an edge e from v_1 to v_2, if $I(e) = S_1 \cup S_2$, where S_1 and S_2 are not empty, then we can split e into two edges as follows: replace e by two edges from v_1 to v_2, one is labeled S_1 and the other is labeled S_2.

5) Subgraphs Pruning: Given two nodes v_1 and v_2, if $F(v_1) = F(v_2)$ and the two subgraphs rooted at v_1 and v_2 are isomorphic, then we can remove v_2 along with the subgraph rooted at v_2, and let all incoming edges of v_2 point to v_1.

6) Subgraphs Replication: If a node v has m ($m \ge 2$) incoming edges, we can replicate m copies of the subgraph rooted at v, and let each incoming edge of v point to a copy.

Operations 2, 4 and 6 are used to make two ordered NDDs semi-isomorphic in this project, while the others are mainly used to reduce an NDD.

A. Node Shaping

Defnition 5.3 (Shapeable Nodes): Let f_a and f_b be two ordered FDDs, v_a be a node of f_a and v_b be a node of f_b. Nodes v_a and v_b are shapeable iff one of the following two conditions holds:

1) Both v_a and v_b have no parents;

2) Both v_a and v_b have parents, the two parents have the same label and the incoming edge of v_a and the incoming edge of v_b have the same label.

For example, the two nodes with label F_1 in figure 6 are shapeable.

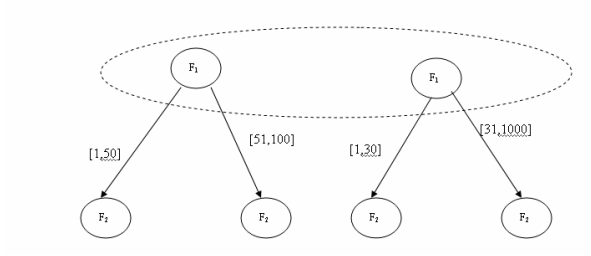

Fig6:Two shapeable nodes before applying the node shaping algorithm.

Definition 5.4 (Semi-isomorphic Nodes): Let f_a and f_b be two ordered NDDs, v_a be a node of f_a

and v_b be a node of f_b. v_a and v_b are semi-isomorphic iff both of the following two conditions hold:

1) v_a and v_b are either both nonterminal nodes with the same label, or both terminal nodes;

2) there exists a one-to-one function σ from the children of v_a to the children of v_b such that for

each child v of v_a, v and $\sigma(v)$ are shapable.

For example, the two nodes with label F_1 in figure 7 are semi-isomorphic.To make two shapable nodes semi-isomorphic, we at first need to make each FDD simple.

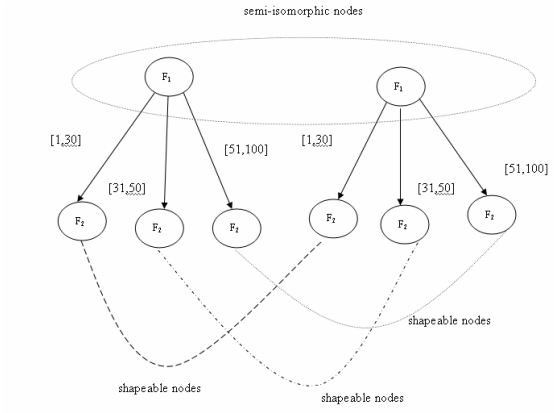

Fig7: Two semi isomorphic nodes after applying node shaping algorithm.

Definition 5.5 (Simple FDDs): An FDD is a simple FDD iff each node has at most one incoming edge and each edge is labelled with a single interval.

A non-simple FDD can be transformed to a simple FDD by the following two steps:

1) *Edge Processing*: Let e be an edge from node v to node v'. Let $I(e) = [a_1,b_1] \cup [a_2,b_2] \cup \ldots \cup [a_m,b_m]$, where $(m \geq 1)$ and for each i, $1 \leq i \leq m\text{--}1$, we have $a_i \leq b_i < a_{i+1}$. We split edge e into m edges $e_1, e_2, \ldots, e_m$, where each e_i is labelled with interval $[a_i, b_i]$. The result of this edge preprocessing is that each edge in is labelled with one single interval.

2) *Node Processing*: For each node v that has more than one incoming edge, we replicate the

subgraph rooted at v into multiple copies, and let each incoming edge of v point to one copy.

The result of node preprocessing is that each node has at most one incoming edge. The algorithm for making two shapeable nodes semi-isomorphic basically consists of two steps:

1) Step I: This step is skipped if v_a and v_b have the same label, or both v_a and v_b are terminal nodes.

Otherwise, without loss of generality, let's assume $F(v_a) < F(v_b)$. Consider the following two cases based on the fact that v_a and v_b are shapeable: (1) Node v_a and v_b have no parents.

Since f_b is an ordered NDD and v_b is its root, in f_b for any node v other than v_b, we have

$F(v_b) < F(v)$ and therefore $F(v_a) < F(v)$. So along all the decision paths containing v_b, no nodes are labelled $F(v_a)$. (2) The two parent nodes of v_a and v_b have the same label. We use p_{v_a} to denote the parent of v_a and p_{v_b} to denote the parent of v_b. Because $F(v_a) < F(v_b)$, $F(p_{v_a}) < F(v_a)$, and $F(p_{v_b}) = F(p_{v_a})$, we have $F(p_{v_b}) < F(v_a) < F(v_b)$. So along all the decision paths containing node v_b, no nodes are labeled $F(v_a)$. In both cases, we can insert a node above v_b with label $F(v_a)$. In other words, create a node v_b' with label $F(v_a)$, create one edge from v_b' to v_b, and the edge is labelled with the domain of $F(v_a)$. Now the root of f_b becomes v_b', and v_a have the same label with v_b'.

2) Step II: Now we can assume that v_a and v_b have the same label. In this step, we use edge splitting and subgraph replication transformations to build a one-to-one correspondence from the children of v_a to the children of v_b such that each child of v_a and its corresponding node are shapeable. Suppose $\Delta F(v_a) = \Delta F(v_b) = [a, b]$. We know each outgoing edge of v_a or v_b is labeled with a single interval. Suppose v_a has k $(k \geq 1)$ outgoing edges $\{e_1, e_2, \ldots e_k\}$ where $I(e_i) = [a_i, b_i]$, $a_1 = a$, $a_{i+1} = b_i + 1$, $b_k = b$ for $1 \leq i \leq k - 1$; and v_b has m $(m \geq 1)$ outgoing edges $\{e_1', e_2', \ldots e_g'\}$, where $I(e_i') = [a_i', b_i']$, $a_1' = a$, $a_{i+1}' = b_{i+1}'$, $b_m' = b$ for $1 \leq i \leq m\text{--}1$. Comparing edge e_1, whose label is $[a, b_1]$, and e_1', whose label is $[a, b_1']$, we have the following two cases: (1) $b_1 = b_1'$: In this case $I(e_1) = I(e_1')$, therefore node $e_1.t$ and node $e_1.t$ are shapeable. Then we can continue to compare e_2 and e_2' since both $I(e_2)$ and $I(e_2')$ begin with $b_1 + 1$. (2) $b_1 \neq b_1'$ without loss of generality, we assume $b_1 < b_1'$. In this case, we split e_1' into two edges, e with label $[a, b_1]$ and e' with label $[b_1 + 1; b_1']$. Then we make two copies of the subgraph rooted at $e_1'.t$ and let e and e' point to one copy each. Thus $I(e_1) = I(e)$ and the two nodes, $e_1.t$ and $e.t$ are shapeable. Then we can

continue to compare edge e_2 and edge e' since both $I(e_2)$ and $I(e')$ begin with $b_1 + 1$.

The above process continues until we reach the last outgoing edge of v_a and the last outgoing edge of v_b. Note that each time when we compare an outgoing edge of v_a and an outgoing edge of v_b, the two intervals labelled on the two edges begins with the same value. Therefore the last two edges that we compare must have the same label because they both ends with b. In other words, this edge splitting and subgraph replication process will terminate. When it terminates, v_a and v_b become semiisomorphic.

In the following detailed algorithm for making two shapable nodes semi-isomorphic, we use $I(e) < I(e')$ to represent that all the integers in $I(e)$ is less than each integer in $I(e')$.

Figure 8 shows the result after we apply this Node Shaping algorithm to the shapeable nodes in figure 7.

Node_Shaping(v_a, v_b)

Input : *Two shapable nodes: v_a of an ordered simple FDD f_a and v_b of an ordered simple FDD f_b.*

Let $E(v_a)$ be $\{e_{a,1}, e_{a,2}, \cdots, e_{a,m}\}$ where $I(e_{a,1}) < I(e_{a,2}) < \cdots < I(e_{a,m})$

Let $E(v_b)$ be $\{e_{b,1}, e_{b,2}, \cdots, e_{b,n}\}$ where $I(e_{b,1}) < I(e_{b,2}) < \cdots < I(e_{b,n})$

Output: *(1) v_a and v_b become semi-isomorphic.*

(2) A set of pairs where each consists of two shapable nodes:

one is a child of v_a and the other is child of v_b.

1. **if** *v_a and v_b are terminal nodes* **then return**($\emptyset$);

   if $F(v_a) \prec F(v_b)$ **then** *insert a new node with label $F(v_a)$ above v_b,*

   and henceforth the new node is called v_b;

   if $F(v_b) \prec F(v_a)$ **then** *insert a new node with label $F(v_b)$ above v_a,*

   and henceforth the new node is called v_a;

2. $i := 1$; $j := 1$;

   while (($i < m$) or ($j < n$)) **do**{

   /*During this loop, the two intervals $I(e_{a,i})$ and $I(e_{b,j})$ always begin with the same integer.*/

   let $I(e_{a,i}) = [A, B]$ and $I(e_{b,j}) = [A, C]$, **where** *A, B, C are three integers;*

   if $B = C$ **then**{

   $i := i + 1$; $j := j + 1$; }

   else if $B < C$ **then**{

   (a) *Create an outgoing edge e of v_b, $I(e) = [A, B]$;*

   (b) *Replicate the graph rooted at $e_{b,j}.t$ and let e point to the replicated graph;*

   (c) $I(e_{b,j}) := [B + 1, C]$;

   (d) $i := i + 1$;}

   else {/*$B > C$*/

   (a) *Create an outgoing edge e of v_a, $I(e) = [A, C]$;*

   (b) *Replicate the graph rooted at $e_{a,j}.t$ and let e point to the replicated graph;*

   (c) $I(e_{a,i}) := [C + 1, B]$;

   (d) $j := j + 1$;}

   }

3. /*Now v_a and v_b become semi-isomorphic.*/

   let $E(v_a) = \{e_{a,1}, e_{a,2}, \cdots, e_{a,k}\}$ **where** $I(e_{a,1}) < I(e_{a,2}) < \cdots < I(e_{a,k})$ and $k \geq 1$;

   let $E(v_b) = \{e_{b,1}, e_{b,2}, \cdots, e_{b,k}\}$ **where** $I(e_{b,1}) < I(e_{b,2}) < \cdots < I(e_{b,k})$ and $k \geq 1$;

   $S := \emptyset$;

   for $i = 1$ **to** k **do add** *a pair of shapable nodes ($e_{a,i}.t$, $e_{b,i}.t$) to S*;

   return(S);

B. FDD Shaping

To make two ordered NDDs f_a and f_b semi-isomorphic, at first we make f_a and f_b simple, then we make f_a and f_b semi-isomorphic as follows. Suppose we have a queue Q, which is initially empty. At first we put the pair of shapable nodes consisting of the root of f_a and the root of f_b into Q. As long as Q is not empty, we remove the head of Q, feed it to the above Node Shaping algorithm, then put all the pairs of shapable nodes returned by the algorithm into Q. Note that all the nodes in the shapable pairs returned by the Node Shaping algorithm consist of all the children of the two nodes we give to the algorithm as input. When the algorithm finishes, the one-to-one correspondence from the nodes of f_a to the nodes of f_b is built. The detail of the shaping algorithm is as follows:

Let the NDD in figure 9 be f_a, and let f_b be the NDD that is as same as f_a except the labels of the black terminal nodes. Here f_a and f_b are the two resulting FDDs

after we apply the above shaping algorithm to the two FDDs, one in figure 3 and one in figure 6. Note that f_a and f_b are semi-isomorphic.

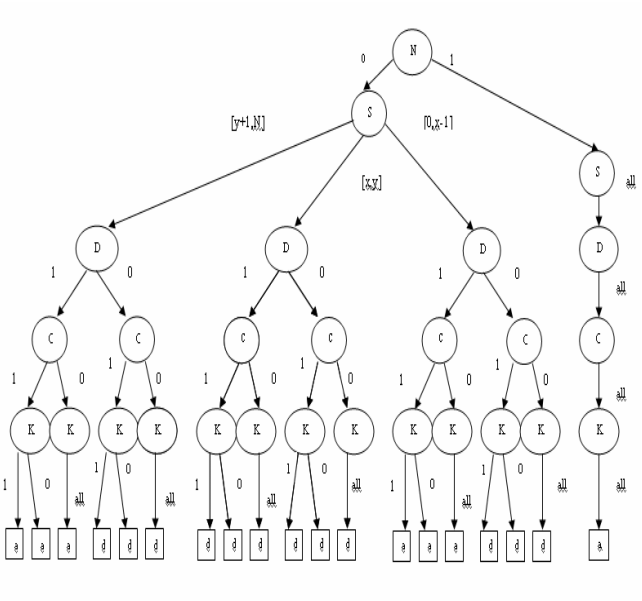

Fig8: NDD transformed from the rule based method.

VI. COMPARISON ALGORITHM

In this section, we consider how to compare two semi-isomorphic NDDs. Given two semi-isomorphic NDDs f and f', for any decision path $(v_1\ e_1\ v_2\ e_2 \ldots v_k\ e_k\ v_{k+1})$ in f, there is a corresponding decision path $(\sigma(v_1)\ \sigma(e_1)\ \sigma(v_2)\ \sigma(e_2) \ldots \sigma(v_k)\ \sigma(e_k)\ \sigma(v_{k+1}))$ in f'. Similarly, for rule $(F(v_1) \in I(e_1)) \cap (F(v_2) \in I(e_2)) \cap \ldots \cap (F(v_k) \in I(e_k)) \rightarrow F(v_{k+1})$ in $S_{f'}$, there is a corresponding rule $(F(\sigma(v_1)) \in I(\sigma(e_1))) \cap (F(\sigma(v_2)) \in I(\sigma(e_2))) \cap \ldots \cap (F(\sigma(v_k)) \in I(\sigma(e_k))) \rightarrow F(\sigma(v_{k+1}))$ in $S_{f'}$ Each of these two rules is called the companion of the other. This companionship implies a one-to-one function from the rules defined by the decision paths in f to the rules defined by the decision paths in f'. By the definition of semi-isomorphic NDDs, for each i, $1 \le i \le k$, we have $I(e_i) = I(\sigma(e_i))$ and $F(v_i) = F(\sigma(v_i))$ because v_i and $\sigma(v_i)$ are nonterminal nodes. But $F(v_{k+1})$ and $F(\sigma(v_{k+1}))$ are not necessarily same because v_{k+1} and $\sigma(v_{k+1})$ are terminal nodes. Therefore the companion of rule $(F(v_1) \in I(e_1)) \cap (F(v_2) \in I(e_2)) \cap \ldots \cap (F(v_k) \in I(e_k)) \rightarrow F(v_{k+1})$ is in fact $(F(v_1) \in I(e_1)) \cap (F(v_2) \in I(e_2)) \cap \ldots \cap (F(v_k) \in I(e_k)) \rightarrow F(\sigma(v_{k+1}))$. For each rule and its companion, either they are identical, or they have the same predicate but different decisions. So S_{f_a} -- S_{f_b} is the set of all the rules in S_{f_a} that have different decisions from their companions. Similarly for S_{f_b} -- S_{f_a}. Since these two sets manifest the discrepancies between the two NDDs, the two design teams can investigate them to resolve the discrepancies.

Let f_a be the NDD in Figure 8, and let f_b be the NDD that is as same as f_a except the labels of the black terminal nodes. Here f_a is equivalent to the network in figure 3 designed by Team A, and f_b is equivalent to the network designed by team B as given in the rule based design section. By comparing f_a and f_b, We discover that there is one discrepancy between the two networks designed by Team A and B. Hence it proves the importance of Diversity based design methods.

VII. CONCLUSIONS

To conclude, in this project we investigated the design diversity problem. First of all we made the taxonomy of different design diversity techniques. Then we applied Construction and shaping methods for Private network model.Finally we are able to find out discrepancy between two solutions.

ACKNOWLEDGMENTS

The authors would like to thank Prof.Miroslaw Malek for providing extremely usefull guidance and his time.

REFERENCES

[1] Yang, Liu and Gouda, "Firewall Correctness by Design Diversity".

[2] [Avizienis 77] Avizienis, A. and L. Chen, "On the implementation of N-version programming for software fault-tolerance during program execution," *Proc. First Intl.Computer Software and Applications Conference*, pp. 149-155, 1977.

[3] [Avizienis 84] Avizienis, A. and J. P. J. Kelly, "Fault Tolerance by Design Diversity: Concepts and Experiments," *IEEE Computer*, pp. 67-80, August, 1984.

[4] [Lala 94] Lala, J. H. and R. E. Harper, "Architectural principles for safety-critical real-time applications," *Proc.of the IEEE*, vol. 82, no. 1, pp. 25-40, January, 1994.

[5] [Riter 95] Riter, R., "Modeling and Testing a Critical Fault-Tolerant Multi-Process System," *Proc. FTCS*, pp.516-521, 1995.

[6] [Tamir 84] Tamir, Y. and C. H. Sequin, "Reducing common mode failures in duplicate modules," *Proc. ICCD*, pp. 302-307, 1984.