FORMAL MODELING OF LINUX-PAM CONFIGURATIONS

FORMALLY SPECIFYING AND VERIFYING LINUX-PAM CONFIGURATIONS USING HIERARCHICAL COLOURED PETRI NETS AND NUSMV

By

CHRISTOPHER KULBAKAS, B.MATH

A Thesis

Submitted to the School of Graduate Studies

in Partial Fulfilment of the Requirements

for the Degree

Master of Science

McMaster University

MASTER OF SCIENCE (2010) McMaster University
(Computing & Software) Hamilton, Ontario

TITLE: Formally Specifying and Verifying Linux-PAM Configurations Using Hierarchical Coloured Petri Nets and NuSMV

AUTHOR: Christopher Kulbakas, B.Math (University of Waterloo)

SUPERVISOR: Professor S. Qiao

NUMBER OF PAGES: xi, 186

Abstract

Authentication frameworks and their implementations are essential to securing computer systems and networks. One such framework is the Pluggable Authentication Modules (PAM) published in 1995 as standard OSF-RFC 86.0. PAM solves the "authentication problem", mainly, how to "integrate multiple authentication mechanisms in a modular and dynamic fashion", making PAM the de facto choice for authentication on most Unix and GNU/Linux-based systems. Linux-PAM is an implementation of PAM for GNU/Linux.

To this day, Linux-PAM configurations are poorly understood by administrators. Ad hoc, informal PAM-configuration testing techniques exist, but suffer from many shortcomings.

We introduce an automated, formal approach to Linux-PAM configuration testing. First, given a Linux-PAM configuration, we dynamically create an "internal-to-the-tool" representation of a Hierarchical Coloured Petri Net (HCPN), which encodes all of the possible authentication process instances associated with this configuration. During this creation, "base case" HCPN templates are used, each template created only once, i.e. during tool development, not during testing. Second, we translate the resulting HCPN into a NuSMV model. Third, we use NuSMV to verify the model for "security" properties.

A tool prototype, implemented in C, automates these three steps. State space size was reduced via manual HCPN and Transition System specification tuning. The State Space Explosion problem was overcome with the use of NuSMV-implemented model checking algorithms. Industrial Linux-PAM configurations were tested, yielding model building times in ones of seconds, and verification times in tens of seconds. Also, the tool produces HCPN representations via Graphviz.

Acknowledgements

I would like to extend my sincere appreciation and gratitude to my supervisor, Dr. Sanzheng Qiao, whose guidance and unending patience has made the completion of this thesis possible.

I would also like to express my gratitude to my committee members, Dr. Ryszard Janicki and Dr. William Farmer, who provided excellent contributions to my study throughout. You also expressed a great amount of patience through these trying times – for this you cannot be thanked enough.

To Dr. Ridha Khedri and Gord – thank you for the discussions and your enthusiasm – it strengthened my sense of purpose and kept me believing. Last, but not least, to Dr. Asghar Bokhari and Issam – thank you for sharing your insights and encouragement.

Finally, I would like to express a heartfelt thank you to my family, who provided me with unending support and encouragement. To Mom, Dad, and Adam, thank you for every form of support you have given me throughout – I only hope that I have made you proud. To Mandy and Luis, you too are to be thanked for your continuous support and for reminding me of my priorities. Lastly, to Monica, *you* were my bright, shiny light in some very dark places. I only hope that I can be the type of partner to you, like you have been to me.

Table of Contents

List of Figures

List of Tables

INTRODUCTION

What is Linux-PAM

Linux-PAM (1), created in 1996 by A. Morgan, is an open source authentication mechanism primarily meant for GNU/Linux systems. Linux-PAM implements the Pluggable AuthenticationModules (PAM) framework.

The PAM framework describes a standardized way to do authentication. The PAM framework was created in 1995 at SunSoft by V. Samar and R. Schemers, and was published in 1995 as an Open Source Foundation (OSF) Request For Comments (RFC) standard OSF-RFC 86.0 (2). The PAM framework's improvements over prior authentication frameworks made it the de facto choice for authentication on most Unix and GNU/Linux-based systems for the past 14 years. To this day, implementations of the PAM framework are the primary authentication technology on a wide variety of platforms including FreeBSD, NetBSD, MAC OS X, Sun Solaris and major GNU/Linux distributions.

The main goals of the PAM framework addressed limitations of existing authentication frameworks (2), (3). These goals included the following. The system administrator should be able to specify the default, system-wide authentication policy, as well as per application authentication policies. Authentication policies should not only address authentication, but also other authentication-related tasks related to account, session and password management. The administrator should be able to integrate the functionality of multiple authentication mechanisms to carry out individual authentication-related tasks.

Last, but not least, changes to authentication policies should not require changes to the applications that use these authentication policies. In other words, the applications were to be made independent of the underlying authentication mechanisms.

The PAM framework solved the unified login problem or "how to integrate multiple authentication mechanisms" (3) which allowed for authentication mechanisms to be field-replacable, and made applications independent of the authentication mechanisms being used. This was made possible by: identifying, abstracting and grouping of tasks associated with authentication; encapsulating the usage of authentication mechanisms into modules; allowing these modules to be dynamically combined to form complex authentication-related functionalities; and providing standardized application programmer's interfaces (APIs) through which applications would make use of these authentication mechanisms.

Pluggable Authentication Modules

The PAM framework introduces the notion of a pluggable authentication module (PAM). PAMs encapsulate authentication mechanism functionality. Each PAM provides some authentication-related functionality, whether on its own, or with the help of external-to-the-PAM authentication mechanisms. For example, in Figure 1 the pam_cracklib.so PAM uses multiple authentication mechanisms including the CrackLib2 package (CrackLib2), an external dictionary, and the underlying operating system to access the external dictionary. A PAM's authentication-related functionality is accessed

via a Module API. For example, pam_cracklib.so PAM's functionality is accessed via pam_sm_chauthtok().

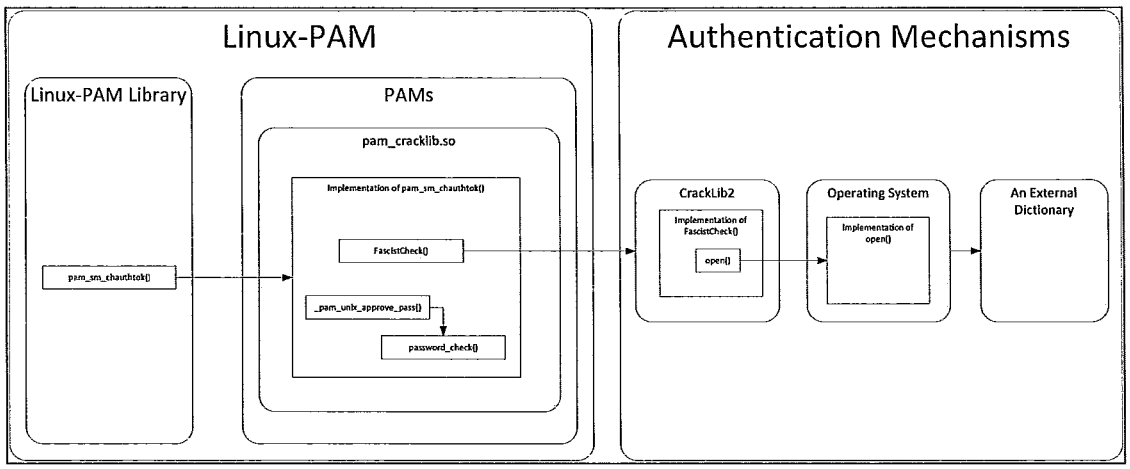

Figure 1: Authentication-related functionality of pam_cracklib.so PAM

Authentication-related functionality varies between PAMs.

Table 1, on page 5, shows this variability for a number of PAMs, partitioned by Module API functions, and by Management Groups (each Module API function is associated with a Management Group).

PAM	Authentication Management		Account Management	Session Management		Password Management
	pam_sm_authenticate()	pam_sm_setcred()	pam_sm_acct_mgmt()	pam_sm_open_session()	pam_sm_close_session()	pam_sm_chauthtok()
pam_securetty.so	check if on secure tty, if user is 'root'	return PAM_SUCCESS	check if on secure tty, if user is 'root'	N/A	N/A	N/A
pam_env.so	return PAM_IGNORE	set environment variables read in from a file	return PAM_SERVICE_ERR	set environment variables read in from a file	return PAM_SUCCESS	return PAM_SERVICE_ERR
pam_unix.so	authenticate user	return PAM_SUCCESS	check user account properties	send log on notification to syslog	send log off notification to syslog	update password of user
pam_succeed_if.so	determine if the supplied condition is true or false	return PAM_IGNORE	determine if the supplied condition is true or false	determine if the supplied condition is true or false	determine if the supplied condition is true or false	determine if the supplied condition is true or false
pam_deny.so	return PAM_AUTH_ERR	return PAM_CRED_ERR	return PAM_AUTH_ERR	return PAM_SESSION_ERR	return PAM_SESSION_ERR	return PAM_AUTHTOK_ERR
pam_nologin.so	return error, if /etc/nologin exists	return PAM_IGNORE	return error, if /etc/nologin exists	N/A	N/A	N/A
pam_permit.so	set username to "nobody", if username is not supplied	return PAM_SUCCESS	return PAM_SUCCESS	return PAM_SUCCESS	return PAM_SUCCESS	return PAM_SUCCESS
pam_cracklib.so	N/A	N/A	N/A	N/A	N/A	check strength of the new, user-supplied password

pam_selinux.so	return PAM_AUTH_ERR	return PAM_SUCCESS	N/A	initialize a session context using the Security Enhanced Linux subsystem	terminate a session context using the Security Enhanced Linux subsystem	N/A
pam_keyinit.so	N/A	N/A	N/A	instantiate a Kernel session keyring using the Kernel Key Retention Service	revoke a Kernel session keyring using the Kernel Key Retention Service	N/A
pam_limits.so	N/A	N/A	N/A	impose resource limits	return PAM_SUCCESS	N/A
pam_loginuid.so	N/A	N/A	sets "Login ID" of Client process in the Linux Audit Subsystem	sets "Login ID" of Client process in the Linux Audit Subsystem	return PAM_SUCCESS	N/A

Table 1: Authentication-Related Functionality of PAMs by Module API function

Table 1 shows PAM authentication-related functionality in a simplified manner. In general, PAM authentication-related functionality is complex in terms of: execution logic, how PAM options affect PAM functionality, user input, state of the underlying operating system and the possible PAM return values. For example, the pam_nologin.so functionality shown in Table 1 only shows default behaviour. To illustrate the complexity of pam_nologin.so PAM's authentication-related functionality, Table 2 shows the execution paths of the pam_nologin.so PAM along with a semantic annotation of the meaning of these execution paths.

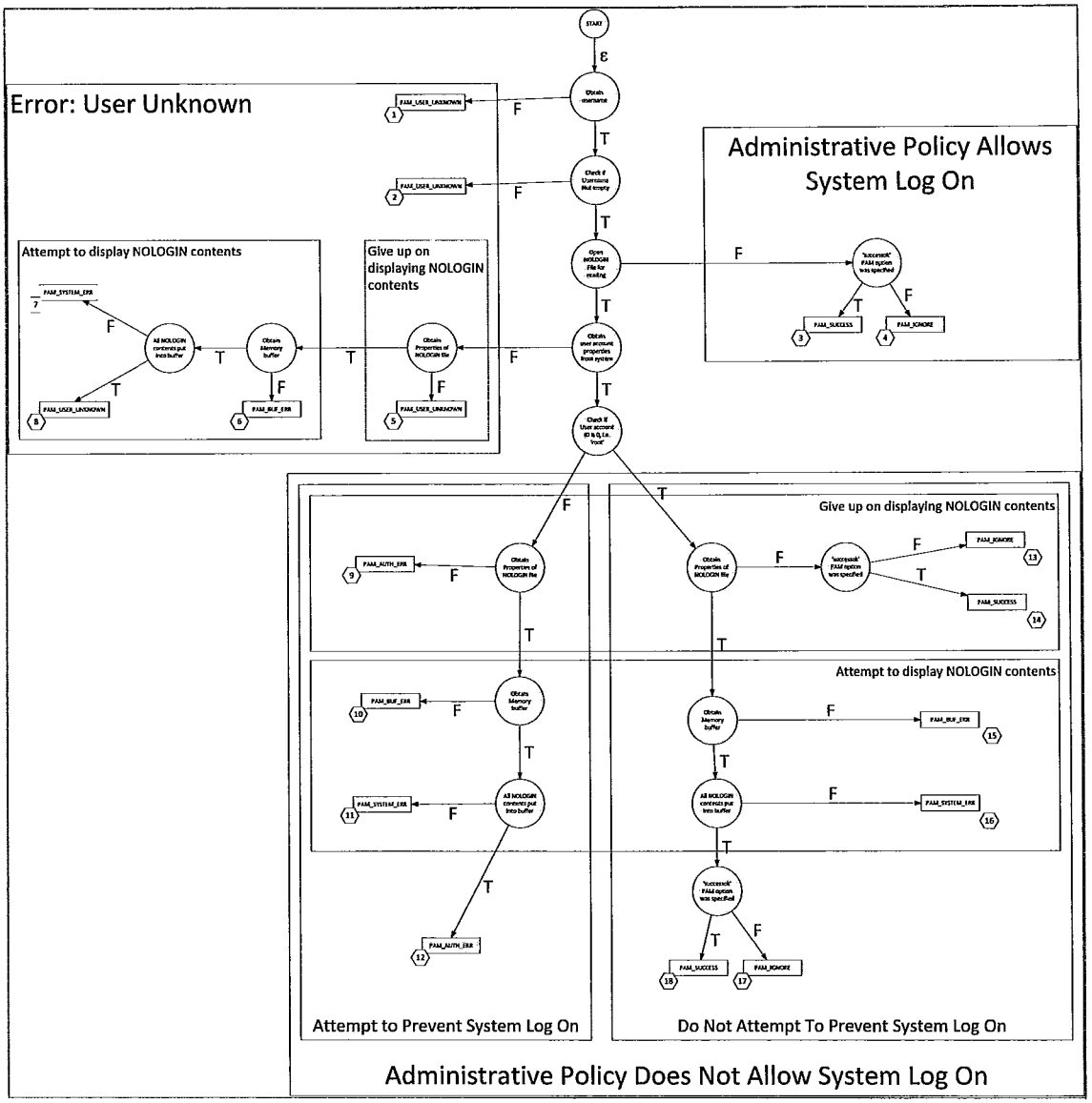

Table 2: Execution tree for pam_nologin.so PAM with semantic annotation

The execution traces, and the factors affecting which execution traces are followed, are summarized in Table 3 .

PAM name	PAM return value	user name obtained	user name not empty	opened NOLOGIN file for reading	obtained user account properties from system	user account ID is 0, i.e. 'root'	obtained properties of NOLOGIN file	obtained memory buffer	put all NOLOGIN contents into buffer	"successok" PAM option specified	execution trace
pam_nologin.so	PAM_USER_UNKNOWN	F	X	X	X	X	X	X	X	X	1
		T	F	X	X	X	X	X	X	X	2
		T	T	T	F	X	F	X	X	X	5
		T	T	T	F	X	T	T	T	X	8
	PAM_IGNORE	T	T	F	X	X	X	X	X	F	3
		T	T	T	T	T	F	X	X	F	13
		T	T	T	T	T	T	T	T	F	17
	PAM_SUCCESS	T	T	F	X	X	X	X	X	T	4
		T	T	T	T	T	F	X	X	T	14
		T	T	T	T	T	T	T	T	T	18
	PAM_BUF_ERR	T	T	T	F	X	T	F	X	X	6
		T	T	T	T	T	T	F	X	X	15
		T	T	T	T	F	T	F	X	X	10
	PAM_SYSTEM_ERR	T	T	T	F	X	T	T	F	X	7
		T	T	T	T	T	T	T	F	X	16
		T	T	T	T	F	T	T	F	X	11
	PAM_AUTH_ERR	T	T	T	T	F	F	X	X	X	9
		T	T	T	T	F	T	T	T	X	12

Table 3: Factors Affecting Return Values of pam_nologin.so PAM - Organized by return value of PAM

Standardized Authentication Interfaces

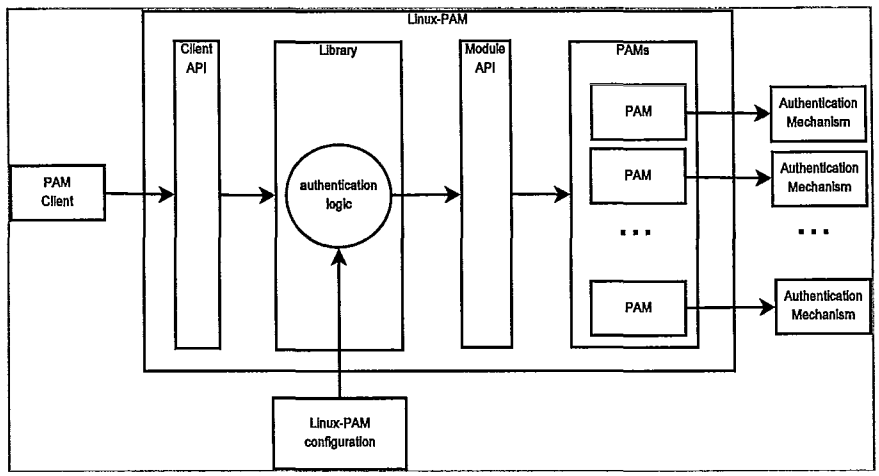

Figure 2: Architecture - Linux-PAM APIs

Along with the use of PAMs, the PAM framework achieves the separation of applications from the underlying authentication mechanisms by using standardized application programmer's interfaces (APIs). This is accomplished by Linux-PAM providing two APIs: an API for applications, called the Client API; and an API for PAMs, called the Module API. The Client API allows applications to use authentication mechanisms without the applications having to contain any mechanism specific programming. The Module API provides a consistent way for authentication mechanism developers to provide access to the mechanism's authentication-related functionality.This is shown in Figure 2.

The Client API and the Module API achieve not only the separation, but also independence of applications, authentication policy, and authentication mechanisms. Any changes in how Linux-PAM or authentication mechanisms operate are transparent to the

Client applications. Any changes in how authentication mechanisms operate are transparent to the Library component of Linux-PAM. If an authentication mechanism changes, it is only the corresponding PAM that may have to be modified. This is because when a PAM is modified, the PAM still has to adhere to the Module API. The Module API is standardized and does not change under any circumstance. All PAM implementations, i.e. Linux-PAM, OpenPAM, Solaris PAM, must implement the Client and the Module APIs.

The Client API is called by Client applications to request authentication-related functionality from Linux-PAM. The Client API consists of the following functions: pam_authenticate(), used for authentication; pam_setcred(), used for setting credentials; pam_acct_mgmt(), used for checking of account status; pam_open_session(), used for opening a log on session; pam_close_session() , used for closing a log on session; and pam_chauthtok(), used for updating of authentication tokens. For example, as shown in Figure 3, an authentication process of the login program makes use of all of the Client API calls.

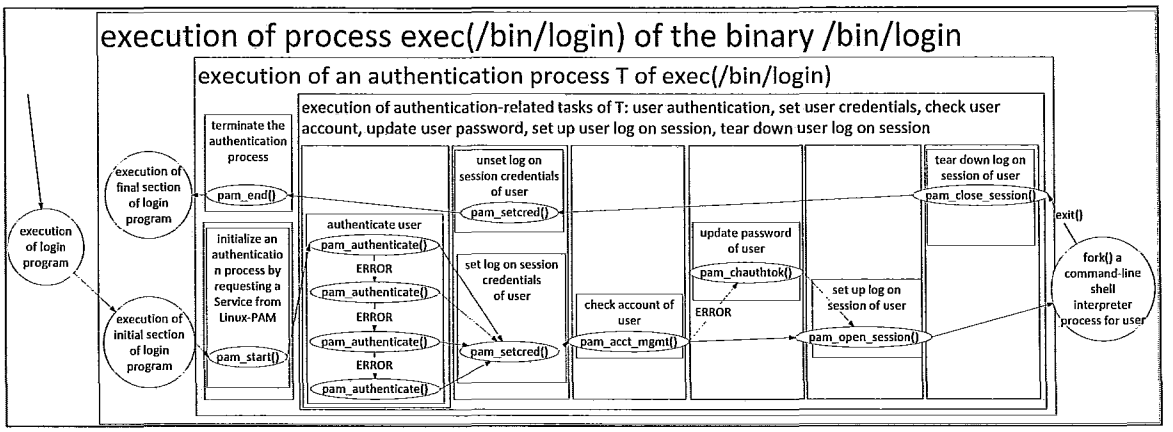

Figure 3: Architecture - Linux-PAM Client Execution

Integration of Authentication Mechanisms via PAM Stacks

Linux-PAM allows for integration of multiple authentication mechanisms. We want to integrate authentication mechanisms in order to provide an aggregate authentication functionality that is a combination of the authentication functionalities of the individual authentication mechanisms (i.e. using pam_cracklib.so PAM to check password strength, and then using pam_unix.so PAM to update the user's password on the system). Integration of authentication mechanisms is achieved using the notion of a PAM Stack.

Each PAM corresponds to some authentication mechanisms. A PAM Stack is a data structure which specifies how some sequence of PAMs can be utilized to provide their authentication-related functionalities.

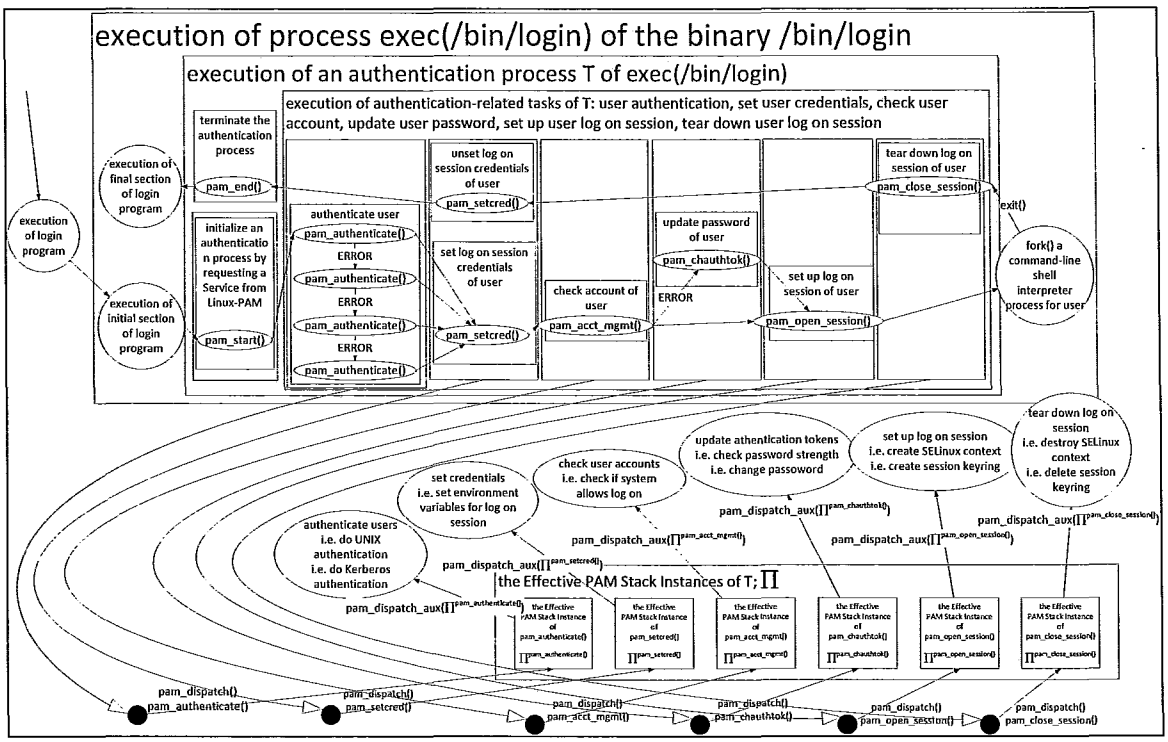

Figure 4: Using Effective PAM Stacks to obtain authentication-related functionality

During an authentication process, Linux-PAM determines a unique PAM Stack instance for each Client API Management Function. The PAM Stack instance is called the Effective PAM Stack instance of the Management Function, for a total of six Effective PAM Stack instances, one Effective PAM Stack instance per Management Function. The Effective PAM Stack instance (of a Management Function) provides some authentication-related functionality for the Management Function, every time the Management Function is called by the Client, for the lifetime of the authentication process. For example, as shown in Figure 4, each time login calls a Management Function, Linux-PAM uses the corresponding Effective PAM Stack instance to provide authentication-related functionality. To illustrate, when login calls pam_chauthtok() to change the user's

password, pam_chauthtok() calls the Linux-PAM Library function pam_dispatch(). Pam_dispatch() identifies the Effective PAM Stack instance corresponding to pam_chauthtok(), denoted $\Pi^{pam\_chauthtok()}$. Then, pam_dispatch() calls pam_dispatch_aux() and provides $\Pi^{pam\_chauthtok()}$ as an argument to pam_dispatch_aux(), denoted pam_dispatch_aux($\Pi^{pam\_chauthtok()}$). Lastly, pam_dispatch_aux() obtains the authentication-related functionality from $\Pi^{pam\_chauthtok()}$. In this case, the authentication-related functionality provided by $\Pi^{pam\_chauthtok()}$ changes the user's password.

For each authentication process, Linux-PAM assigns two types of Linux-PAM configurations to a client: client-specific and default. At the start of an authentication process, a Linux-PAM Client requests from PAM to use a Service S, i.e. Service named "login", i.e. Service named "sshd". In turn Linux-PAM identifies a unique file, called a root file, as the starting point of the Linux-PAM configuration, for each of the client-specific and the default configuration types. The identification of these root files is not trivial, as shown in Table 4.

	Exists /etc/pam.d/	Exists /etc/pam.d/SERVICE_NAME	Exists /etc/pam.d/other	PAM_READ_BOTH_CONFS	Exists /etc/pam.conf	find_cs(SERVICE_NAME) find_df('other')	parse_rule(find_cs(SERVICE_NAME)) parse_rule(find_df('other'))
1	F	X	X	X	F	ERROR	
2	F	X	X	X	T	/etc/pam.conf	1st token is SERVICE
						/etc/pam.conf	1st token is SERVICE
3	T	F	F	X	X	ERROR	
4	T	T	T	X	X	/etc/pam.d/SERVICE_NAME	1st token is GROUP
						/etc/pam.d/other	1st token is GROUP
5	T	T	F	X	X	/etc/pam.d/SERVICE_NAME	1st token is GROUP
						UNDEFINED	N/A
6	T	F	T	T	T	/etc/pam.conf	1st token is SERVICE
						/etc/pam.d/other	1st token is GROUP
7	T	F	T	T	F	UNDEFINED	N/A
						/etc/pam.d/other	1st token is GROUP
8	T	F	T	F	X	UNDEFINED	N/A
						/etc/pam.d/other	1st token is GROUP

Table 4: Algorithm for Identification of the Root of a Service Configuration Specification and the Parsing Rule of the Root

Once identified, Linux-PAM parses the root files, and generates four PAM Stack specifications for each of client-specific and default configurations, where each of the four specifications correspond to a distinct Management Group. Then, Linux-PAM generates six PAM Stack instances (one for each Client API Management Function) based on the four specifications. Given a PAM Stack specification for a Management Group, this specification is used to generate the PAM Stack instances for all Management Functions belonging to this group, i.e. PAM Stack specification for Authentication Management generates two PAM Stack instances: one for pam_authenticate() and one for pam_setcred(). Again, this is done for each of the client-specific and the default configurations. Thus, twelve PAM Stack instances are created in total. From these twelve, using each Client API Management Function, the client-specific and default PAM Stack instances are paired up, and an Effective PAM Stack instance is chosen. If the client-specific PAM Stack instance has no "stacked" PAMs configured, then the default PAM Stack instance is used. This process is summarized in Figure 5, by using the login program as an example.

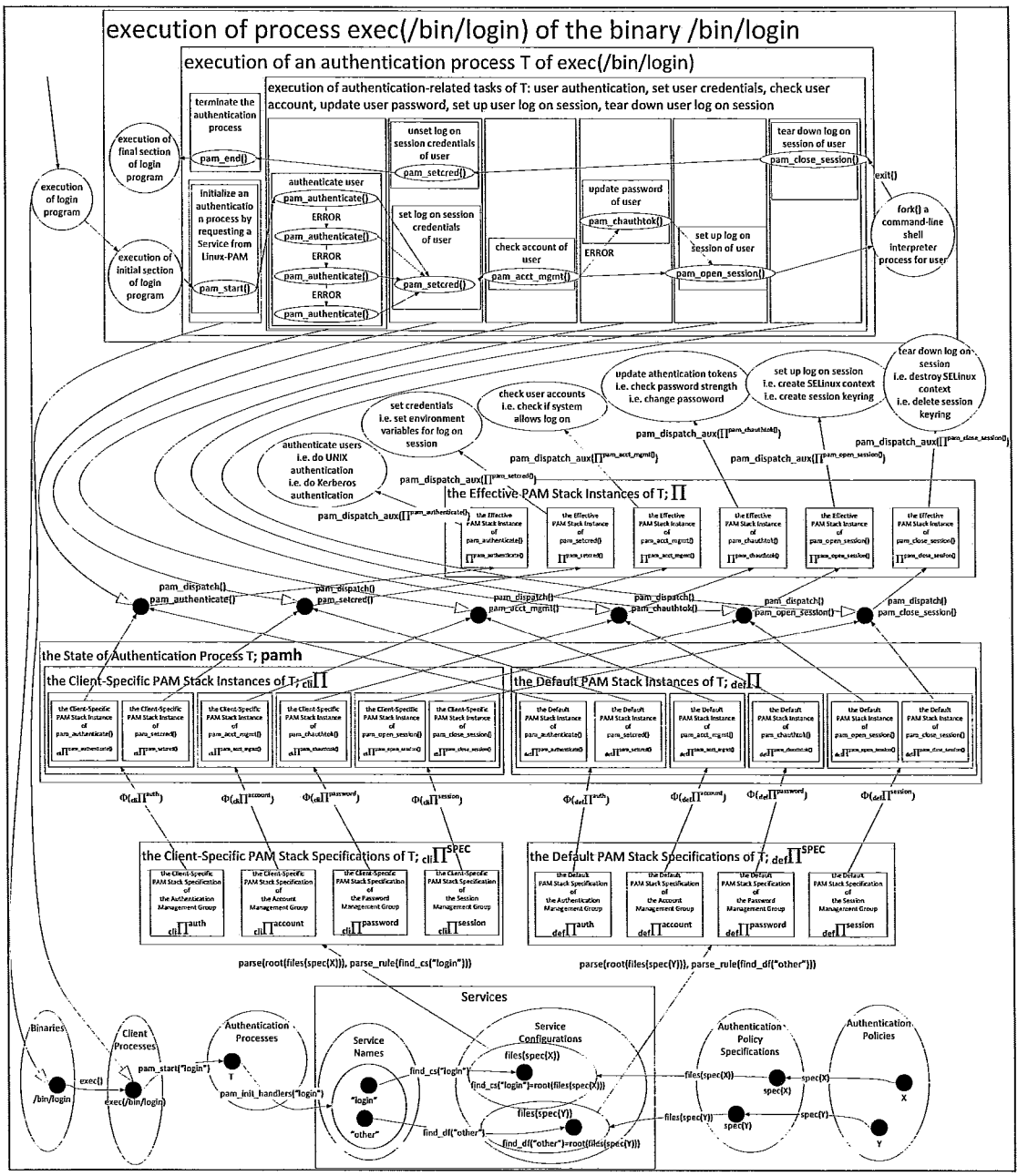

Figure 5: High Level Overview of Effective PAM Stack creation for, and usage by, an Authentication Process

The next section describes PAM Stack composition and functionality.

PAM Stack Composition and Functionality

Introduction

The PAM framework achieves the integration of the functionality of multiple authentication mechanisms using the notion of a PAM stack.

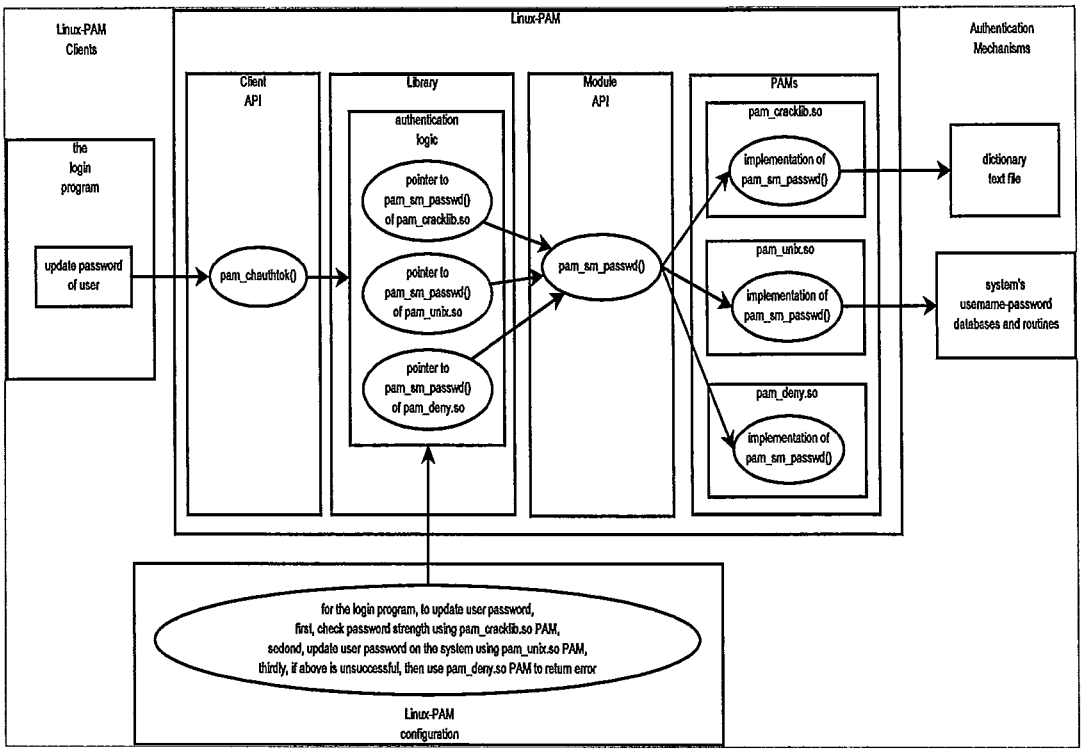

Figure 6: Using APIs to update a password

First, by example, we show how the Client and Module APIs can be used to carry out an authentication-related task of updating of a password. Figure 6 shows the architecture of the system used in updating of this password. The sequence of events describing this password update is listed in Figure 7. This sequence of events occurs when the login program updates the user's password via Linux-PAM.

1. Supposing that the Administrator supplies the following Configuration for updating of user authentication tokens for the login program: a. the strength of the newly proposed password must be checked via the pam_cracklib.so PAM, then, if successful; b. update the user's password on the system via the pam_unix.so PAM, then, if not successful; c. use the pam_deny.so PAM to return an error.
2. The login Client calls the (Client API) pam_passwd() to change the user's password.
3. In turn, the Library calls the (Module API) pam_sm_passwd() implementation of each of the PAMs, as dictated by the Configuration, in this case: a. calling pam_cracklib.so PAM's implementation of pam_sm_passwd(); b. followed by calling pam_unix.so PAM's implementation of pam_sm_passwd(), if the above was successful; c. followed by calling pam_deny.so PAM's implementation of pam_sm_passwd(), if the above was unsuccessful.
4. When pam_sm_passwd() of pam_cracklib.so PAM is called, its execution: a. provides pam_cracklib.so PAM's authentication-related functionality for updating of authentication tokens, mainly, checking of the password strength, b. makes use of the actual authentication mechanism that corresponds to this PAM, in this case, pam_cracklib.so uses internal password check routines, and can optionally use external dictionaries.
5. When pam_sm_passwd() of pam_unix.so PAM is called, its execution: a. provides pam_unix.so PAM's authentication-related functionality for updating of authentication tokens, mainly updating the system's password database with the new password information; b. makes use of the actual authentication mechanism that corresponds to this PAM, in this case, pam_unix.so may use passwd and shadow system password databases and routines.
6. When pam_sm_passwd() of pam_deny.so PAM is called, its execution: a. provides pam_deny.so PAM's authentication-related functionality for updating of authentication tokens, mainly, returning an error; b. does not make use of an actual authentication mechanism - this PAM does not have a corresponding authentication mechanism.

Figure 7: Using APIs to update a password - event sequence

There is a 1-1 correspondence between Client API functions and Module API functions, i.e. pam_authenticate() corresponds to pam_sm_authenticate(). The Client uses Client API calls to request service for authentication-related tasks from Linux-PAM. In turn, the Library component of Linux-PAM uses the corresponding Module API calls to request the functionality for the authentication-related task from each individual PAM

being used. In a proper configuration, each PAM belonging to a PAM Stack of some Client API function must implement the corresponding Module API function. We say that a PAM is *stacked* on a PAM Stack, if the PAM Stack uses the PAM to provide authentication related functionality.

```
struct handler {
    int handler_type;
    int (*func)(pam_handle_t *pamh, int flags, int argc, char **argv);
    int actions[_PAM_RETURN_VALUES];
    int cached_retval;
    int *cached_retval_p;
    int argc;
    char **argv;
    struct handler *next;
    char *mod_name;
    int stack_level;
};
```

Table 5: Definition of 'struct handler' Data Structure

Which, and how, the stacked PAMs can be used by a PAM Stack instance, is encoded in each PAM Stack Instance. On the implementation level, PAM Stack Instances are linked lists of struct handler data structures (Table 5) which contain memory addresses of the Module API function implementations (int (*func)), which we denote by the symbol $I(f^{sm}, P)$. Handlers also store an encoding of all of the possible ways that these function pointers can be invoked to execute some subsequence of the Module API function implementations comprising the PAM Stack Instance (int actions[_PAM_RETURN_VALUES]).

Which PAMs are stored on a PAM Stack Instance, and in what sequences these PAMs may be used, is defined by the Linux-PAM Administrator in a Linux-PAM Configuration. Before we present the procedure used by Linux-PAM to generate an authentication-

related functionality using a PAM Stack Instance, we first show how a PAM Stack Instance is generated based on a Linux-PAM Configuration.

Generating a PAM Stack Instance from a Linux-PAM Configuration

A Linux-PAM Configuration, for the Linux-PAM Service login (Table 6, Table 7), is used to generate a PAM Stack Specification for the Authentication Management group (Table 8)[1]. In Table 9 we show an intermediate step where we transform each Complex Controls of each configuration line into the Control Action definition using names of Actions, for each "stacked" PAM. In Table 10 we show an intermediate step where we transform each Complex Control of each configuration line into the Control Action definition using numbers (not names) of PAM_RETURN values and Actions, for each "stacked" PAM. This last form is then used to generate a condensed form of the corresponding PAM Stack Specification for pam_authenticate() (Table 11). By condensed, we refer to simplifying the description of the Control Function by generating a sequence of partitions of PAM_RETURN = {0,1,...,31}, and using the keyword "ow", using the algorithm outlined in Table 12. We use this final form of the PAM Stack Instance for generation of HCPN models of PAM Stack Instance executions.

[1] Due to a brevity requirement, the procedure for this, and subsequent examples, was omitted.

```
login auth [user_unknown=ignore success=ok ignore=ignore default=bad]
pam_securetty.so
login auth       include      system-auth

login account    required     pam_nologin.so
login account    include      system-auth

login password   include      system-auth

login session    required     pam_selinux.so close
login session    include      system-auth
login session    required     pam_loginuid.so
login session    optional     pam_console.so
login session    required     pam_selinux.so open
login session    optional     pam_keyinit.so force revoke
```

Table 6: Start of Client-Specific portion of Linux-PAM Configuration for Service 'login'

```
auth        required      pam_env.so
auth        sufficient    pam_unix.so nullok try_first_pass
auth        requisite     pam_succeed_if.so uid >= 500 quiet
auth        required      pam_deny.so

account     required      pam_unix.so
account     sufficient    pam_succeed_if.so uid < 500 quiet
account     required      pam_permit.so

password    requisite     pam_cracklib.so try_first_pass retry=3
password    sufficient    pam_unix.so md5 shadow nullok try_first_pass use_authtok
password    required      pam_deny.so

session     optional      pam_keyinit.so revoke
session     required      pam_limits.so
session     [success=1 default=ignore] pam_succeed_if.so service in crond quiet use_uid
session     required      pam_unix.so
```

Table 7: Linux-PAM Configuration in the /etc/pam.d/system-auth File

depth	level	SERVICE S	SERVICE GROUP	CONTROL C:{0,1,...,31}->ACTIONS	PATH P	OPTIONS O
0	0	login	auth	[user_unknown=ignore success=ok ignore=ignore default=bad]	pam_securetty.so	
1	0	login	auth	[success=ok new_auth_tok_reqd = ok ignore=ignore default=bad]	pam_env.so	
2	0	login	auth	[success=done new_auth_tok_reqd = ok default=ignore]	pam_unix.so	nullok try_first_pass
3	0	login	auth	[success=ok new_auth_tok_reqd = ok ignore=ignore default=die]	pam_succeed.so	uid >= 500 quiet
4	0	login	auth	[success=ok new_auth_tok_reqd = ok ignore=ignore default=bad]	pam_deny.so	

Table 8: Generation of PAM Stack Specification for the Authentication Management group for Service "login" with Complex Control equivalents

Depth i		0	1	2	3	4
PATH P_i		pam_ securetty .so	pam_ env .so	pam_ unix .so	pam_ succeed_if .so	pam_ deny .so
CONTROL C_i	PAM_SUCCESS	ok	ok	done	ok	ok
	PAM_OPEN_ERR	bad	bad	ignore	die	bad
	PAM_SYMBOL_ERR	bad	bad	ignore	die	bad
	PAM_SERVICE_ERR	bad	bad	ignore	die	bad
	PAM_SYSTEM_ERR	bad	bad	ignore	die	bad
	PAM_BUF_ERR	bad	bad	ignore	die	bad
	PAM_PERM_DENIED	bad	bad	ignore	die	bad
	PAM_AUTH_ERR	bad	bad	ignore	die	bad
	PAM_CRED_INSUFFICIENT	bad	bad	ignore	die	bad
	PAM_AUTHINFO_UNAVAIL	bad	bad	ignore	die	bad
	PAM_USER_UNKNOWN	ignore	bad	ignore	die	bad
	PAM_MAXTRIES	bad	bad	ignore	die	bad
	PAM_NEW_AUTHTOK_REQD	bad	ok	ok	ok	ok
	PAM_ACCT_EXPIRED	bad	bad	ignore	die	bad
	PAM_SESSION_ERR	bad	bad	ignore	die	bad
	PAM_CRED_UNAVAIL	bad	bad	ignore	die	bad
	PAM_CRED_EXPIRED	bad	bad	ignore	die	bad
	PAM_CRED_ERR	bad	bad	ignore	die	bad
	PAM_NO_MODULE_DATA	bad	bad	ignore	die	bad
	PAM_CONV_ERR	bad	bad	ignore	die	bad
	PAM_AUTHTOK_ERR	bad	bad	ignore	die	bad
	PAM_AUTHTOK_RECOVERY_ERR	bad	bad	ignore	die	bad
	PAM_AUTHTOK_LOCK_BUSY	bad	bad	ignore	die	bad
	PAM_AUTHTOK_DISABLE_AGING	bad	bad	ignore	die	bad
	PAM_TRY_AGAIN	bad	bad	ignore	die	bad
	PAM_IGNORE	ignore	ignore	ignore	ignore	ignore
	PAM_ABORT	bad	bad	ignore	die	bad
	PAM_AUTHTOK_EXPIRED	bad	bad	ignore	die	bad
	PAM_MODULE_UNKNOWN	bad	bad	ignore	die	bad
	PAM_BAD_ITEM	bad	bad	ignore	die	bad
	PAM_CONV_AGAIN	bad	bad	ignore	die	bad
	PAM_INCOMPLETE	bad	bad	ignore	die	bad
OPTIONS O_i				nullok	uid	
				try_first _pass	>=	
					500	
					quiet	
LEVEL L_i		0	0	0	0	0

Table 9: PAM Stack Specification ${}^{T}_{cli}\Pi^{\text{auth}}_{\text{login}}$ with Control Function definitions

Depth i		0	1	2	3	4
PATH P_i		pam_ securetty .so	pam_ env .so	pam_ unix .so	pam_ succeed_if .so	pam_ deny .so
CONTROL C_i	0	-1	-1	-2	-1	-1
	1	-3	-3	0	-4	-3
	2	-3	-3	0	-4	-3
	3	-3	-3	0	-4	-3
	4	-3	-3	0	-4	-3
	5	-3	-3	0	-4	-3
	6	-3	-3	0	-4	-3
	7	-3	-3	0	-4	-3
	8	-3	-3	0	-4	-3
	9	-3	-3	0	-4	-3
	10	0	-3	0	-4	-3
	11	-3	-3	0	-4	-3
	12	-3	-1	-1	-1	-1
	13	-3	-3	0	-4	-3
	14	-3	-3	0	-4	-3
	15	-3	-3	0	-4	-3
	16	-3	-3	0	-4	-3
	17	-3	-3	0	-4	-3
	18	-3	-3	0	-4	-3
	19	-3	-3	0	-4	-3
	20	-3	-3	0	-4	-3
	21	-3	-3	0	-4	-3
	22	-3	-3	0	-4	-3
	23	-3	-3	0	-4	-3
	24	-3	-3	0	-4	-3
	25	0	0	0	0	0
	26	-3	-3	0	-4	-3
	27	-3	-3	0	-4	-3
	28	-3	-3	0	-4	-3
	29	-3	-3	0	-4	-3
	30	-3	-3	0	-4	-3
	31	-3	-3	0	-4	-3
OPTIONS O_i				nullok	uid	
				try_first_pass	>=	
					500	
					quiet	
LEVEL L_i		0	0	0	0	0

Table 10: PAM Stack Specification ${}^{T}_{cli}\Pi^{\text{auth}}_{\text{login}}$ with Control Function definitions in "raw" format

depth; i		*level;* L_i	*SERVICE;* S_i	*SERVICE GROUP;* G_i	*CONTROL;* C_i			*PATH;* P_i	*OPTIONS;* O_i
					$X \subseteq R(C_i)$	$d \in D(C_i)$: $\forall x \in X$: $C_i(x) = d$			
	int handler_type	int stack_level			int actions[32]		int (*func)	char *mod_name	char **argv
0	0	0	login	auth	0	-1	I(pam_sm_authenticate(), pam_securetty.so)	pam_securetty	
					10,25	0			
					"ow"	-3			
1	0	0	login	auth	0,12	-1	I(pam_sm_authenticate(), pam_env.so)	pam_env	
					25	0			
					"ow"	-3			
2	0	0	login	auth	0	-2	I(pam_sm_authenticate(), pam_unix.so)	pam_unix	nullok
					12	-1			try_first_pass
					"ow"	0			
3	0	0	login	auth	0,12	-1	I(pam_sm_authenticate(), pam_succeed.so)	pam_succeed	uid
					25	0			>=
					"ow"	-4			500
									quiet
4	0	0	login	auth	0,12	-1	I(pam_sm_authenticate(), pam_deny.so)	pam_deny	
					25	0			
					"ow"	-3			

Table 11: Generation of a PAM Stack Instance for the pam_sm_authenticate() Module API function for theService "login"

Given Table 11, the mapping, from the table column labeled $X \subseteq R(C_i)$, to the table column labeled $d \in D(C_i): \forall x \in X: C_i(x) = d$, is determined by steps listed in Table 12.

Input: Control Function $C: \{0,1,\ldots,31\} -> \{-5,-4,\ldots,0,\ldots 65535\}$, Control Function is total
1. Group domain of C into partitions of {0,1,...,31} by the images being mapped to. 2. Remove 31 from the partition containing it. 3. Sort partitions by minimum element, in increasing order. 4. Move partition with most elements and largest minimal member to the last row and denote by "ow".
Output: sequence of partitions $\{X\}$ s.t. $\forall$ terms $X \subseteq R(C): \forall x \in X: C(x) = d$.

Table 12: Partitioning of the range of the Control function

Generating of Authentication-Related Functionality from a PAM Stack Instance

Given a PAM Stack instance, Linux-PAM executes this PAM Stack instance in order to generate an instance of authentication-related functionality. This authentication-related functionality is a combination of the authentication-related functionalities of the PAMs comprising this PAM Stack instance. Specifically, Linux-PAM uses the algorithm shown in Table 13 to use the stacked PAMs of the PAM Stack instance to execute the Module API function implementations of these "stacked" PAMs to provide this authentication-related functionality.

Given the PAM Stack Instance in Table 11, this instance has 5 stacked PAMs. Each PAM stacking is implemented using the struct handler data structure shown in Table 5 on

page 19. Hence, we view a PAM Stack instance as a sequence of handlers, i.e. handler_0, handler_1, ..., handler_x-1, where the amount of stacked PAMs is x, and each handler contains a Module API function implementation (precisely, a pointer to this implementation) of some PAM, and a Control Function definition.

INPUT: ${}^{T}\Pi_S^f = \{\text{handler}_i\} = \text{handler}_0, \text{handler}_1, \ldots, \text{handler}_{x-1}$, an Effective PAM Stack Instance with x handlers OUTPUT: pam_return, i.e. a PAM Stack Instance execution return value
1. If $x = 0$, i.e. PAM Stack Instance is empty (i.e. no "stacked" PAMs) a. $pam\_return := PAM\_SYSTEM\_ERROR = 4$ b. go to Step 6
2. Initialize PAM Stack and Substack Execution states by executing initialization code a. $(I, M) := (0,6)$, i.e. initialize PAM Stack Execution State b. $(I_0, M_0) := (0,6)$, i.e. initialize PAM 0^{th} Substack State c. $(I_L, M_L) := (0,0),\ 1 \leq L \leq 15$, i.e. initialize all other PAM Substack States
3. Choose the first handler, $handler_0$, as the first handler to be processed a. $handler_i := handler_0$
4. While the chosen $handler_i, 0 \leq i \leq x-1$, exists on the PAM Stack instance (is a term of $\{\text{handler}_i\}$), process $handler_i$, denoted $exec(handler_i)$, where last(i) denotes the last handler depth to be processed a. if $level(handler_{last(i)}) < level(handler_i) = K$, i.e. entering higher substack level K, then i. $(I_K, M_K) := (I, M)$, i.e. save current PAM Stack Execution State b. If $type(handler_i) = PAM\_HT\_MUST\_FAIL$, i.e. handler is erroneous, then i. $return_i := PAM\_MUST\_FAIL\_CODE = 6$ ii. go to step 4(g), i.e. skip PAM execution, go directly to execution of Action c. If $type(handler_i) = PAM\_HT\_SUBSTACK$, i.e. start of Substack, then i. $i := i + 1$, i.e. choose next handler in sequence to be executed ii. go to step 4 d. If $\nexists\, I(f^{sm}, P_i)$, i.e. PAM does not implement the Management Function i. $return_i := PAM\_MODULE\_UNKNOWN = 28$ ii. go to step 4(g), i.e. skip PAM execution, go directly to execution of Action e. $exec(I(f^{sm}, P_i), O_i)$, i.e. execute the implementation of Module API function f^{sm} of PAM P_i i. $return_i := exec(I(f^{sm}, P_i), O_i)$, i.e. obtain result from execution of $I(f^{sm}, P_i)$ f. If $return_i = PAM\_INCOMPLETE = 31$, i.e. PAM needs more information from user in order to provide anuthentication-related functionality i. $pam\_return := return_i = PAM\_INCOMPLETE = 31$ ii. go to step 6 g. $action_i := C_i(return_i)$, i.e. determine the mapped-to Action $action_i$ h. $exec(action_i, return_i, (I, M))$, i.e. execute $action_i$ i. $((I, M), i) := exec(action_i, return_i, (I, M))$, i.e. update PAM Stack Execution State and choose next handler to execute ii. go the Step 4
5. Terminate PAM Stack execution state a. return PAM Stack execution result pam_return by executing termination code $terminate()$, denoted $exec(terminate())$ i. $pam\_return := exec(terminate((I, M))$
6. Return result of PAM Stack Instance Execution a. return pam_return

Table 13: High-level Overview of PAM Stack Execution

As shown in Table 13 the authentication-related functionality of a PAM Stack Instance Execution is comprised of a subsequence of handler executions. Each handler execution, under non-substack and non-erroneous circumstances, consists of executing the Module API function implementation contained in the handler. The choice of the next handler to be executed depends on the current PAM Stack Execution state. The final PAM_RETURN value determines whether or not the PAM Stack execution is deemed as successful. A return value of PAM_SUCCESS = 0 means that the PAM Stack execution was successful. Otherwise, PAM Stack execution was not successful. Note that, whether or not the PAM Stack execution is deemed as successful still does not change the fact that the PAM Stack execution has already provided authentication-related functionality via its handler executions. The final PAM_RETURN value has utility for the Linux-PAM Client, but not for Linux-PAM.

The main idea behind PAM Stack execution is as follows. PAM Stack execution starts with the first handler. Then, in the case of a non-erroneous stacked PAM, the PAM's implementation of the Module API function is executed to provide some authentication-related functionality. Then, the PAM_RETURN value obtained from this Module API function execution is used by the Control function to choose an Action to execute. Execution of an Action does two things: first, the PAM Stack Execution State is updated; second, the next handler to be executed is chosen. The updating of the PAM Stack Execution state, as well as the choosing of the next handler may be dependent on the

current PAM Stack Execution state, as well as the latest PAM_RETURN value obtained from the Module API function execution[2].

The particular way in which each Action executes is not described in this section. Rather, we utilize our HCPN encodings to show how these Actions operate.

The execution of handlers during a PAM Stack execution is a subsequence of the handler sequence comprising the PAM Stack Instance.

PAM Stack execution stops when Termination execution is reached. At this point, a decision regarding the final PAM Stack execution return value is made by the Termination procedure. This decision is based on the current PAM Stack Execution State, and is described by the state machine shown in Figure 8.

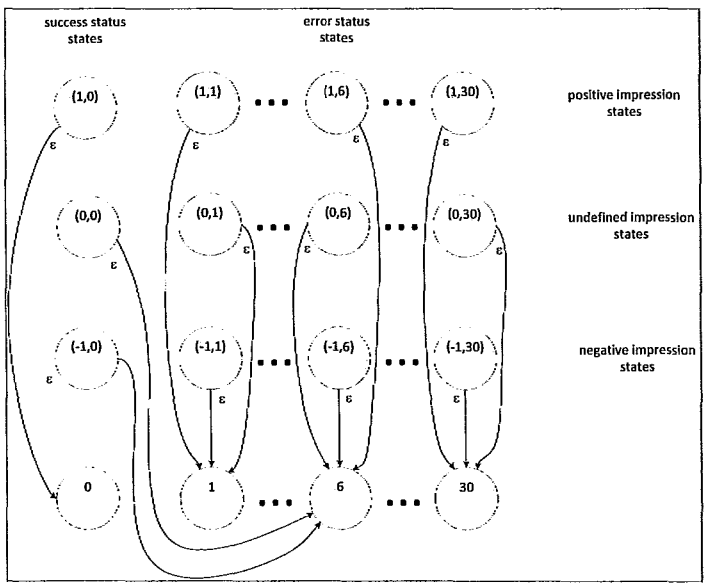

Figure 8: FSM of PAM Stack Dispatch State and Depth Under Termination

[2] It may also be dependent on previous return values obtained for the same Management Function call, just not for the same Management Function call instance, but we do not model this as this is the Frozen Chain functionality. Frozen Chain is part of future work.

It is the goal of this thesis, and of the pamtester-fm tool, to model all possible ways in which a PAM Stack Instance can be executed, where this PAM Stack Instance is generated based on an arbitrary Linux-PAM Configuration.

The next section delves into our HCPN modeling.

METHODOLOGY - PART I: HCPN Modeling

Introduction to HCPN Modeling

Current Petri Net software tools are limited. For example, these tools do not allow programmatic, dynamic generation of arbitrary Petri Nets, and programmatic control, such as automated simulation. Due to this, as part of this thesis work, we developed a software tool called pamtester-fm. The purpose of pamtester-fm is to create HCPN structures of arbitrary PAM Stack Instance executions, via automated, programmatic means.

HCPN models encode, not only structure of a system, but also a system's behaviour. In particular, our HCPN models of PAM Stack Instance executions not only encode the structure of a PAM Stack Instance execution, but also its "executable behaviour".

The problem in our case is that, although an HCPN encodes behaviour of a system, one still needs to somehow simulate or execute this HCPN behaviour . Otherwise, the HCPN specification is just a textual specification, not something dynamic or executable. In analogy, one can specify some "execution" using the C programming language, but the resulting program still has to be compiled and executed to analyze the program's run-time behaviour. Thus, additionally, pamtester-fm makes use of an external program called NuSMV(4) to simulate the HCPN's behaviour.

From Source Code to HCPN

Our creation of HCPN models is based on the knowledge of how Linux-PAM would process an Effective PAM Stack's linked list of struct handlers to obtain its authentication-related functionality.

This knowledge is based on a source code audit of the files implementing the functions that obtain authentication-related functionality of PAM Stacks, mainly, each of the Client API functions: pam_start(), pam_authenticate(), pam_setcred(), pam_open_session(), pam_close_session(), pam_chauthtok(), the pam_dispatch() function, and finally, the pam_dispatch_aux() function.

For example, a typical call chain for a PAM Stack Instance execution is as follows. First, a Linux-PAM Client establishes an authentication process with Linux-PAM. This is done by the Client calling pam_start() and supplying the name of the Linux-PAM Service as an argument. In turn, Linux-PAM uses the procedure described in Figure 5 on page 16, to generate the set of client-specific and default PAM Stack Instances. Once generated, Linux-PAM provides these PAM Stack Instances to the Linux-PAM Client. Then, the Linux-PAM Client makes a sequence of Linux-PAM API Management Function calls. Each time one of these function calls is made, the Client supplies the PAM Stack Instances as arguments. Thus, Linux-PAM obtains the PAM Stack Instances from the Client, every time the Client calls a Management Function. Now, suppose that the Client called pam_authenticate(). The source code for pam_authenticate() is shown below.

```
int pam_authenticate(pam_handle_t *pamh, int flags)
{
    int retval;

    D(("pam_authenticate called"));

    IF_NO_PAMH("pam_authenticate", pamh, PAM_SYSTEM_ERR);

    if (__PAM_FROM_MODULE(pamh)) {
  D(("called from module!?"));
  return PAM_SYSTEM_ERR;
    }

    if (pamh->former.choice == PAM_NOT_STACKED) {
  _pam_sanitize(pamh);
  _pam_start_timer(pamh);    /* we try to make the time for a failure
              independent of the time it takes to
              fail */
    }

    retval = _pam_dispatch(pamh, flags, PAM_AUTHENTICATE);

    if (retval != PAM_INCOMPLETE) {
  _pam_sanitize(pamh);
  _pam_await_timer(pamh, retval);   /* if unsuccessful then wait now */
  D(("pam_authenticate exit"));
    } else {
  D(("will resume when ready"));
    }

#ifdef PRELUDE
    prelude_send_alert(pamh, retval);
#endif

#ifdef HAVE_LIBAUDIT
    retval = _pam_auditlog(pamh, PAM_AUTHENTICATE, retval, flags);
#endif

    return retval;
}
```

Figure 9: pam_authenticate – source code

Here, we abstract everything but the call to execute the Effective PAM Stack Instance (Figure 10).

```
retval = _pam_dispatch(pamh, PAM_AUTHENTICATE);.
```

Figure 10: Pam_authenticate() – resulting abstracted source code to be modeled by HCPN

The first line, for example, is a debug call (Figure 11). We do not need to model it.

```
D(("pam_authenticate called"));
```

Figure 11: Pam_authenticate() – debug call – not modeled

The next line (Figure 12) implements a call that checks if the data structure containing the PAM Stack Instances (provided to Linux-PAM by the Client) is NULL (empty). This is simply a check that "protects" Linux-PAM from rouge, or improperly implemented Linux-PAM Client applications, or faulty PAMs, or even Linux-PAM bugs. Essentially, here, Linux-PAM is protecting itself from any piece of code that would set the memory address, reserved for pointing to the PAM Stack Instance data structures, to zero (NULL). Again, we abstract this away from our model – we assume this does not happen in our model.

```
IF_NO_PAMH("pam_authenticate", pamh, PAM_SYSTEM_ERR);
```

Figure 12: pam_authenticate() - source code - NULL check not modeled

The next line (Figure 13) is the same idea: here Linux-PAM is making sure that it was not a PAM that made a call to pam_authenticate(). We abstract this away by assuming this will not happen.

```
  if (__PAM_FROM_MODULE(pamh)) {
D(("called from module!?"));
return PAM_SYSTEM_ERR;
  }
```

Figure 13: pam_authenticate() – source code - call source check - not modeled

The next piece of code (Figure 14) starts a Linux-PAM "timer". This piece of code is irrelevant to our problem. We abstract it away by assuming that, in our model, it will always work as expected, without affecting the PAM Stack Instance execution.

```
    if (pamh->former.choice == PAM_NOT_STACKED) {
  _pam_sanitize(pamh);
  _pam_start_timer(pamh);     /* we try to make the time for a failure
                independent of the time it takes to
                fail */
    }
```

Figure 14: pam_authenticate() - source code - timer - not modeled

The lines of code after the PAM Stack dispatch (Figure 10) are not relevant to our modeling of the PAM Stack Instance Execution. Thus, we ignore or abstract away the rest of the code in this file (Figure 15).

```
    if (retval != PAM_INCOMPLETE) {
  _pam_sanitize(pamh);
  _pam_await_timer(pamh, retval);    /* if unsuccessful then wait now */
  D(("pam_authenticate exit"));
    } else {
  D(("will resume when ready"));
    }

#ifdef PRELUDE
    prelude_send_alert(pamh, retval);
#endif

#ifdef HAVE_LIBAUDIT
    retval = _pam_auditlog(pamh, PAM_AUTHENTICATE, retval, flags);
#endif

    return retval;
}
```

Figure 15: pam_authenticate() - source code - post PAM Stack execution - not modeled

Essentially, we are left with the single call to the function that does the PAM Stack Execution (Figure 10).

Following this approach, we determined that we can exercise a significant amount of abstraction when modeling, yet still obtain meaningful results that approximate the functionality of Linux-PAM for a subset of the Client API Management Functions. Specifically, currently, our model approximates the execution of the functions: pam_authenticate(), pam_acct_mgmt() and pam_open_session().

Because of the amount of abstraction used, the current model is not capable of modeling the rest of the Management Functions, mainly: pam_setcred(), pam_close_session() and pam_chauthtok(). This is because our model does not model Frozen Chain – a functionality of Linux-PAM that is a factor in the operation of pam_setcred(), pam_close_session() and pam_chauthtok(). Modeling of Frozen Chain is included as part of future work.

Approach to HCPN Modeling of PAM Stack Executions

In our modeling approach, we model the PAM Stack Execution, as implemented by pam_dispatch() and pam_dispatch_aux() functions. These functions are implemented in the libpam/pam_dispatch.c file in the Linux-PAM source code (1). By using an abstraction approach, as above, and by abstracting certain functionality such as Frozen Chain, passing of flags, and passing of PAM options (see Limitations) we were able to create an abstracted model of PAM Stack Instance Execution, as implemented by these two files.

Specifically, given the level of abstraction used, the relevant portions of pam_dispatch() execution (see Figure 89 on page 176 for complete source code) consist of choosing the effective PAM Stack Instance, given the called Management Function (Figure 16), and then calling pam_dispatch_aux() to initiate the execution of the chosen Effective PAM Stack Instance (Figure 17). All other code portions of pam_dispatch.c were abstracted away using the same approach as for pam_authenticate().

```
    switch (choice) {
     case PAM_AUTHENTICATE:
   h = pamh->handlers.conf.authenticate;
   break;
     case PAM_SETCRED:
   h = pamh->handlers.conf.setcred;
   use_cached_chain = _PAM_MAY_BE_FROZEN;
   break;
     case PAM_ACCOUNT:
   h = pamh->handlers.conf.acct_mgmt;
   break;
     case PAM_OPEN_SESSION:
   h = pamh->handlers.conf.open_session;
   break;
     case PAM_CLOSE_SESSION:
   h = pamh->handlers.conf.close_session;
   use_cached_chain = _PAM_MAY_BE_FROZEN;
   break;
     case PAM_CHAUTHTOK:
   h = pamh->handlers.conf.chauthtok;
   if (flags & PAM_UPDATE_AUTHTOK) {
       use_cached_chain = _PAM_MUST_BE_FROZEN;
   }
...
     if (h == NULL) {      /* there was no handlers.conf... entry; will
use
          * handlers.other... */
   switch (choice) {
   case PAM_AUTHENTICATE:
       h = pamh->handlers.other.authenticate;
       break;
   case PAM_SETCRED:
       h = pamh->handlers.other.setcred;
       break;
   case PAM_ACCOUNT:
       h = pamh->handlers.other.acct_mgmt;
       break;
   case PAM_OPEN_SESSION:
       h = pamh->handlers.other.open_session;
       break;
   case PAM_CLOSE_SESSION:
       h = pamh->handlers.other.close_session;
       break;
   case PAM_CHAUTHTOK:
       h = pamh->handlers.other.chauthtok;
       break;
   }
     }
```

Figure 16: PAM Stack Execution – pam_dispatch() - Choosing the Effective PAM Stack Instance

```
   retval = _pam_dispatch_aux(pamh, flags, h, resumed, use_cached_chain);
```

Figure 17: PAM Stack Execution - pam_dispatch() - Initiating PAM Stack Execution

In our approach, we decided to let pamtester-fm carry out the functionality of choosing the Effective PAM Stack Instance (Figure 16), simply because this process is fixed. In other words, the choosing of the Effective PAM Stack does not vary. Thus, we do not need to include it as part of the model, if we have the opportunity to compute this beforehand. Hence, the choosing of the Effective PAM Stack Instance is implemented directly in pamtester-fm source code.

In contrast, execution of PAM Stack Instances, as implemented by pam_dispatch_aux() (see Figure 90 on 181 for complete source code) is modeled using HCPNs. This modeling is done by describing each of the intermediate steps of the process described in the PAM Stack Execution algorithm (Table 13, page 29). This algorithm describes an abstraction of the PAM Stack Instance execution, as implemented by pam_dispatch_aux().This abstraction is done using the same approach as for pam_authenticate() and pam_dispatch().

Since we are using HCPNs as the modeling language, we can choose the level of abstraction at which to model, using a component-wise, hierarchical approach. Our current model does sufficient modeling of each of the intermediate components so that meaningful modeling simulation of PAM Stack executions can be obtained. The goal is to build a "modeling foundation" on which more detailed, multi-hierarchical models can be built. This is outlined as part of future work.

The HCPN models of the individual elements are then combined together using HCPN constructs such as Substitute Transitions and Fusion Places. This combining of individual HCPNs results in a final HCPN. This final HCPN describes a PAM Stack execution

process of a particular PAM Stack Instance, as a whole. The goal is for this resulting, final HCPN to model all possible PAM Stack Instance executions, and hence all possible authentication-related functionalities of the corresponding PAM Stack Instance.

An HCPN encodes both the structure and the behaviour of a system. In our case, our HCPNs encode the structure, and the behaviour of PAM Stack executions of arbitrary (but fixed, once the model is generated) PAM Stack Instances. The HCPN structural aspect describes both, the components of a PAM Stack Instance execution and the factors that may have impact on the execution of the PAM Stack Instance, as well as the way that Linux-PAM executes PAM Stack Instances. Some examples of structural aspects include: handler sequence, Control function of each handler, which Actions can be executed by processing of each handler, substack level structure, user input variations (i.e. user password is empty), configuration variations (i.e. pam_nologin.so PAM configured to return PAM_SUCCESS, not PAM_ERROR), system memory errors (i.e. memory buffer allocation routine fails), underlying operating system properties (i.e. user account does not exist on the system), etc. The HCPN behavioural aspect describes the possible authentication-related functionalities that can be generated by the PAM Stack Instance. Some examples of behavioural aspects include: possible handler processing sequences, or what happens when the user supplies an empty password vs. a non-empty password. The behavioural aspect is described using the HCPN Firing Rule. The Firing Rule is a mechanism by which we specify how an HCPN may change state. In other words, it encodes the possible behaviours of the HCPN. The Firing Rule, based on the structural

description, encodes all of the possible ways that the corresponding PAM Stack Instance can provide authentication-related functionality.

Finding the Balance in HCPN Encoding

In developing the method for HCPN encoding, we had to address three goals: human readability, HCPN simplicity, and HCPN representation power. On one hand, we strove to make the HCPN as simple as possible. In order to do this, we strove to minimize the amount of Petri Net constructs such as Places and Transitions, as well as their relationships via the Firing Rule. On the other hand we had to ensure that our HCPNs contain the appropriate representation power, in terms of being able to represent the behaviour of PAM Stack Instance executions in a valid manner. Last, but not least, our goal was also to make the HCPN as "human-friendly" and intuitive as possible. In particular, we wanted the HCPN structure to reflect PAM Stack Instance execution's component structure and execution behaviour in a way that was intuitive. Specifically, we wanted HCPN specifications to visually reflect inter-component relationships (i.e. Actions depend on Controls, i.e. Controls depend on Handlers, etc.), and flow of data (i.e. PAM_RETURN values are obtained from Module executions).

The above goals may compete with each other. This is because, at some point, the simpler, or more complex, the HCPN structure, the less intuitive and human-friendly the HCPN representation becomes.HCPN representation power may also be affected. Hence a balance had to be reached between these competing goals.

HCPN Model Specification

Given a PAM Stack Instance

${}^{T}\Pi_S^f = \{\text{handler}_i\} = \text{handler}_0, \text{handler}_1, \ldots, \text{handler}_{n-1}$ containing n handlers, we generate an HCPN model to simulate all of the possible PAM Stack Execution traces of this PAM Stack Instance.

The generation of HCPN models is done automatically by our custom-developed tool called pamtester-fm. Pamtester-fm obtains a PAM Stack Instance from Linux-PAM, parses this PAM Stack Instance, and based on this parsing, generates an HCPN model representing the structure and behaviour of the possible authentication-related functionalities (or PAM Stack Executions) of this PAM Stack Instance.

Our HCPN model of a PAM Stack Instance Execution is composed of multiple HCPN modules. These HCPN modules are combined in a hierarchical fashion, using substitution transitions, to form the HCPN model. We also make use of Fusion Sets to ensure that the visual representation of our HCPN modules is "human-friendly" in terms of two-dimensional graphical representation.

Our HCPN model is generated based on templates. Each HCPN module has a corresponding template. Each template is implemented as a computer program source code making up our pamtester-fm tool. In other words, HCPN module templates are embedded as part of the pamtester-fm tool.

Pamtester-fm tool generates HCPN module specifications based on these templates. These HCPN module specifications are stored in pamtester-fm process memory during the runtime of pamtester-fm.

These HCPN module specifications, in turn, are used by pamtester-fm to generate HPCN module instances. These HCPN module instances comprise the overall HPCN model instance. This HCPN model instance is stored in pamtester-fm process memory during the runtime of pamtester-fm.

Based on this HCPN model instance, pamtester-fm generates the Transition System encoding in NuSMV syntax. This Transition System describes the behaviour of the HCPN model instance in terms of the firing rule. In other words, this Transition System describes how the HCPN model instance can change state. This NuSMV syntax is stored in a text file by pamtester-fm.

Additionally, based on this HCPN model instance, pamtester-fm outputs a two-dimensional graphical representation of the HCPN model instance. To do so, pamtester-fm encodes the visual structure of the HCPN model instance (two-dimensional layout of places, transitions, edges, etc.) using GraphViz syntax. Then, pamtester-fm uses GraphViz to render this structure producing a two-dimensional graphical representation of the HCPN model instance. This graphical representation of the HCPN model instance is a "partial" unfolding of the HCPN model instance. By "partial" we mean that we do not completely follow the steps comprising an HCPN model instance unfolding as specified in (5). Specifically, we do not merge all Fusion Sets, because this would make the graphical representation "too messy".

For the rest of this section, unless noted otherwise, we use the PAM Stack Instance ${}^{\tau}\Pi^{\text{pam\_authenticate()}}_{\text{login}}$ shown in Figure 94 on page 185 to illustrate our examples. This

PAM Stack instance enforces user authentication at ACME Corp for the Linux-PAM Service "login".

HCPN module DISPATCH

We start the modeling of ${}^{T}\Pi_{S}^{f}$ with the HCPN module DISPATCH. DISPATCH is a *prime module*, and is the only prime module in our Petri Net model.

The pamtester-fm HCPN module DISPATCH template is shown in Figure 18.

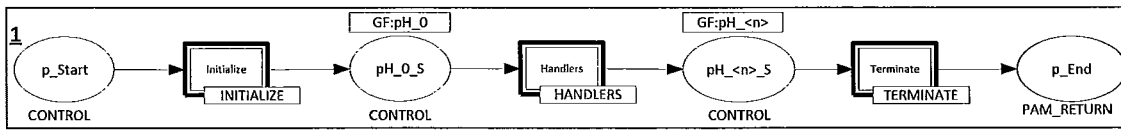

Figure 18: Pamtester-fm HCPN module DISPATCH template

Due to the generic nature of HCPN templates, in order for us to illustrate the structure of HCPN templates, we make use of functions and variables within the renderings of these templates. Specifically, the text contained within the angled brackets '<' and '>' denotes functions and variables. During operation of pamtester-fm, depending on the PAM Stack instance being parsed, the functions and variables contained in these templates are computed accordingly by pamtester-fm. For example, given the place pH_<n>_S in Figure 18, the HCPN module DISPATCH specification for the PAM Stack instance, shown in Figure 94 on page 185, is replaced with 4, i.e. pH_<n>_S := pH_4_S. Number 4 denotes the number of handlers comprising this particular PAM Stack instance.

Based on the template in Figure 18, pamtester-fm generates the HCPN module specification show in Figure 19. Figure 19 shows a rendering of the HCPN module Dispatch specification for user authentication via "login" at ACME Corp.

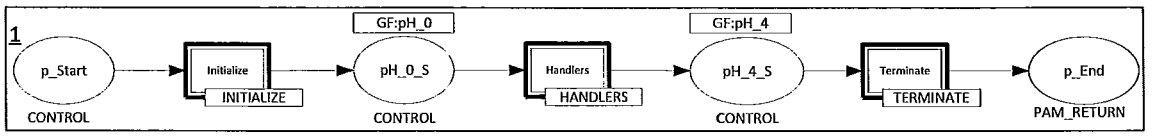

Figure 19: HCPN module DISPATCH specification for authentication via "login" at ACME Corp.

The Dispatch module specification in Figure 19 contains four places: p_Start, pH_0_S, pH_4_S and p_End. Place pH_0_S belongs to the GF:pH_0 fusion set. Place pH_4_S belongs to the GF:pH_4 fusion set. This specification also contains three substitution transitions: Initialize, Handlers and Terminate. p_Start, pH_0_S, pH_4_S can only contain tokens of color CONTROL. Place p_End can only contain tokens of color PAM_RETURN.

The color CONTROL is only used for defining how the HCPN model should "execute". For example, a CONTROL token in place p_Start indicates that the INITIALIZE submodule can start executing. An arrival of a token in place p_Start represents the start of the PAM Stack execution. Precisely, it implicitly denotes that step 1 of the PAM Stack Execution algorithm specified in Table 13 on page 29 was evaluated to false, meaning that the PAM Stack instance contains at least one handler[3]. Secondly, it denotes that the initialization of the PAM Stack and Substack Execution can now be

[3] In fact, pamtester-fm does not generate an HCPN model for a PAM Stack Instance, if this Instance does not contain any handlers. In this case, pamtester-fm simply returns the PAM_RETURN value of _PAM_SYSTEM_ERR = 4, as implemented in the pam_dispatch_aux() function in libpam/pam_dispatch.c

carried out (step 2 of this algorithm). As another example, a CONTROL token in place pH_0_S indicates that the HANDLERS submodule can start executing. Specifically, an arrival of a token in place pH_0_S represents the choosing of the first handler, handler_0, for execution and that this handler can now be executed (step 3 of this algorithm). As another example, a CONTROL token in place pH_4_S indicates that the TERMINATE submodule can start executing. An arrival of a token in pH_4_S represents the end of the PAM Stack execution's handler processing, and that PAM Stack execution's termination code can now be executed (step 5 of this algorithm).

In contrast, the color PAM_RETURN is not only used for defining HCPN execution control, but also for specifying a possible PAM Stack execution result. For example, when a place p_End receives a token color PAM_RETURN, execution-wise, this indicates the finishing of PAM Stack execution. At the same time, this token also represents the returning of the PAM Stack execution result. The value of this token is used as the return value of the PAM Stack execution (step 6a in the algorithm).

Combining HCPN Modules

Thus, the HCPN module DISPATCH specification is the starting point for the overall specification of our HCPN model of a PAM Stack execution. Our HCPN model is comprised of not only the HCPN module DISPATCH. In fact, there are other HCPN modules that are used in the specification of our model. As mentioned above, to construct the overall HCPN model of PAM Stack execution, we combine the HCPN module

DISPATCH with other HCPN modules. We do this combining via substitution transitions.

In the HCPN model specification (and pamtester-fm template), a substitution transition is specified by a rectangular box with a double-lined border. The text contained in the center of the substitution transition is the name of the substitution transition. HCPN module DISPATCH has three substitution transitions: Initialize, Handlers, and Terminate. Each substitution transition corresponds to an HCPN module, indicated by the name of the module being specified in the small textbox found at the bottom of the substitution transition. For example, substitution transition Initialize corresponds to the HCPN module INITIALIZE, Handlers corresponds to HANDLERS, and Terminate corresponds to TERMINATE.

Substitution transitions specify combining of HCPN modules. In the case of the HCPN module DISPATCH specification, this specification dictates that DISPATCH is combined with INITIALIZE, HANDLERS and TERMINATE. The resulting HCPN is composed of the individual HCPNs: DISPATCH, INITIALIZE, HANDLERS and TERMINATE. The structure and behaviour of the resulting HCPN is a combination of the structure and behaviour of the individual HCPNs being used in the creation of the resulting HCPN. In our model, the resulting HCPN's structure and behaviour is a combination of the structure and behaviour of the HCPN modules DISPATCH, INITIALIZE, HANDLERS and TERMINATE.

HCPN modules INITIALIZE, HANDLERS and TERMINATE are called submodules of the HCPN module DISPATCH. We call DISPATCH a parent module of INITIALIZE,

HANDLERS and TERMINATE. A submodule may have other submodules. Thus, HCPN models specify sequences of submodule specifications. For example, since INITIALIZE is a submodule of DISPATCH, denoted DISPATCH → INITIALIZE, thus DISPATCH → INITIALIZE is an example of a sequence of submodule specifications. Given an HCPN model, any submodule sequences of this model must not contain a cycle. For example, if the HCPN module INITIALIZE contained a substitution transition whose corresponding HCPN module was DISPATCH, then we would obtain a sequence of submodule specifications which contains cycles. Such a sequence would have the form DISPATCH → INITIALIZE → DISPATCH → INITIALIZE → …, where a cycle is DISPATCH → INITIALIZE → DISPATCH, for example. We cannot have any such cycles in HCPN specifications as this would result in an infinite HCPN model instance.

HCPN model instances are generated from HCPN model specifications. Using an analogy, an HCPN model specification is a "blueprint", and an HCPN model instance is a "structure" generated based on this "blueprint".

Pamtester-fm generates an HCPN model instance based on the HCPN model specifications.

HCPN module Instances and the Instance Hierarchy

Once an HCPN model specification is obtained, i.e. all HCPN modules comprising this model are specified, the next step is to generate an HCPN instance of this model, based on the HCPN model specification.

We use HCPN instances to study the HCPN model's behaviour. For example, we may be interested in how the HCPN model changes its *marking* – a distribution of tokens amongst the HCPN's places. A sequence of HCPN token marking changes is an HCPN *execution*. Specifically, in the case of modeling PAM Stack instances, we may want to know what possible PAM_RETURN values the particular PAM Stack instance may return. In this case, we use an HCPN model template to generate the appropriate HCPN model specification, based on the PAM Stack instance obtained from Linux-PAM. Then, based on the HCPN model specification, we generate the HCPN model instance. Then, we use the HCPN model instance to simulate the possible executions of the HCPN model.

HCPN model instantiation causes the HCPN module specifications, comprising the HCPN model specification, to be instantiated. This results in an HCPN model instance (comprised of the individual HCPN module instances).

Once we obtain an HCPN model instance, we can simulate HCPN executions. Given an HCPN model instance corresponding to a PAM Stack Instance, we can attempt to enumerate HCPN model instance executions of interest. An execution of our HCPN model instance represents a portion of the PAM Stack instance's execution. An HCPN model instance execution represents a portion of an authentication-related functionality, this functionality as defined by the PAM Stack Execution algorithm (Table 13, page 29).

HCPN model instantiation instantiates all modules, following the submodule sequence. In our model, the HCPN module DISPATCH is instantiated first. Figure 20 shows this instantiation and its *initial marking* – the starting marking. Our initial marking has a single (1') CONTROL token of value 1 in place p_Start (grey rectangle containing 1'1).

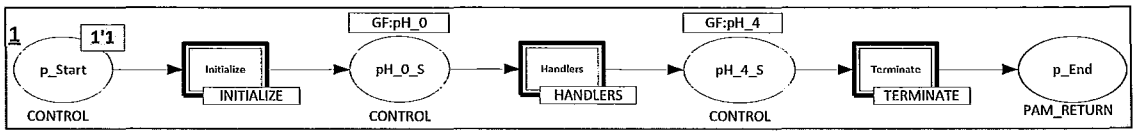

Figure 20: HCPN module DISPATCH specification for authentication via "login" at ACME Corp.

When an HCPN module is instantiated, for each substitution transition of this module, the corresponding HCPN module (submodule) is also instantiated. The instantiated submodules are "combined" with their corresponding parent module. In our model, HCPN module DISPATCH has three substitution transitions called Initialize, Handlers and Terminate. Initialize, Handlers and Terminate correspond to the submodules INITIALIZE, HANDLERS and TERMINATE, respectively. When DISPATCH is instantiated, INITIALIZE, HANDLERS and TERMINATE are also instantiated. In turn, the submodules of INITIALIZE, HANDLERS and TERMINATE cause instantiations of their submodules, and so on, and so forth. For a detailed description of how HCPN instantiation takes place, refer to (5).

Thus, the set of modules of an HCPN model has a binary relation, which we call an instantiation relation. This instantiation relation, represented as a directed graph, forms a set of connected, acyclic subgraphs. A node of the graph is an HCPN module comprising the HCPN model. A directed edge of the graph exists between two nodes A and B, where the edge is outgoing from node A and incoming to another[4] node B, if, and only if, B is a submodule of A. This graph is called an *instance hierarchy* (5). Since our model contains

[4] A is distinct from B

a single prime module, an instance hierarchy of an HCPN model of a PAM Stack Instance is a connected, acyclic graph (a tree).

Instance hierarchies help us visualize how modules and their submodules are combined to create the overall HCPN model instance.

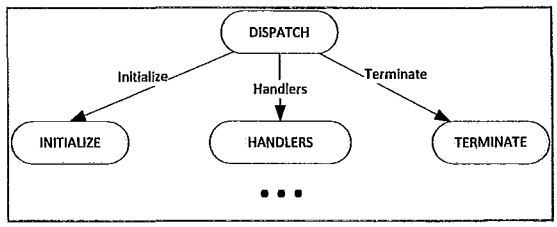

Figure 21: Partial instance hierarchy rooted at DISPATCH

In Figure 21 we show a "partial" instance hierarchy corresponding to the HCPN Dispatch instance shown in Figure 20. As discussed, DISPATCH has three submodules: INITIALIZE, HANDLERS and TERMINATE. Figure 21 shows this fact by the existence of the directed edges, each edge outgoing from DISPATCH, and incoming into each of the corresponding submodules of DISPATCH. The edge labels indicate names of the corresponding substitution transitions. This instance hierarchy is "partial" because we did not show the whole instance hierarchy yet. Specifically, the HCPN module HANDLERS has HCPN submodules which we have not shown yet.

HCPN module INITIALIZE

INITIALIZE models the initialization of the PAM Stack Execution. The pamtester-fm template for the HCPN module INITIALIZE is shown in Figure 22. The corresponding instance is shown in Figure 23.

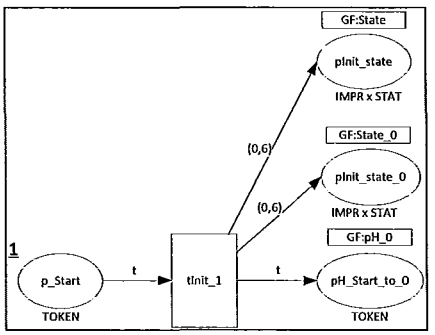

Figure 22: Pamtester-fm HCPN module INITIALIZE template

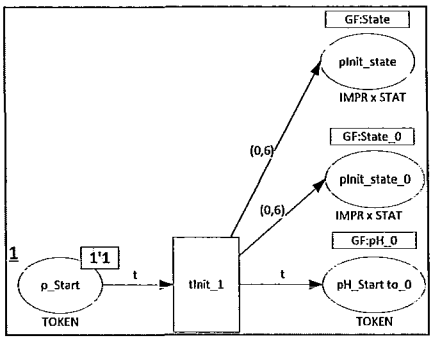

Figure 23: HCPN module INITIALIZE instance; for ACME Corp.

The HCPN module INITIALIZE models the PAM Stack execution initialization. This is the second step in the execution of the PAM Stack instance, as exhibited by the algorithm in Table 13 on page 29. The transition tInit_1, and only transition tInit_1, becomes enabled at initialization, and hence is the only transition that can be chosen to be fired by the HCPN Firing Rule. Once tInit_1 is fired, it consumes a CONTROL token from HCPN place p_Start, sets the PAM Stack Execution state to (0,6), the PAM Substack Level 0 state to (0,6), and places a CONTROL token in the HCPN place pH_Start_to_0. The placing of the CONTROL token in pH_Start_to_0 models the act of choosing the first handler, handler_0, to be executed next.

HCPN module TERMINATE

TERMINATE models the termination of the PAM Stack Execution. The HCPN module TERMINATE template is shown in Figure 24. The HCPN Termination instance is shown in Figure 25.

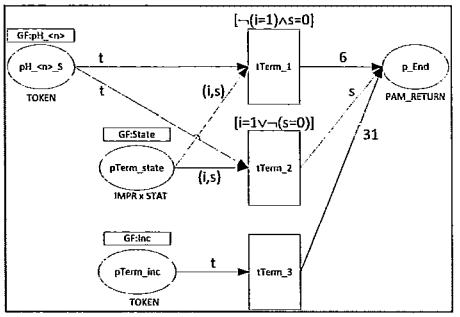

Figure 24: Pamtester-fm HCPN module TERMINATE template

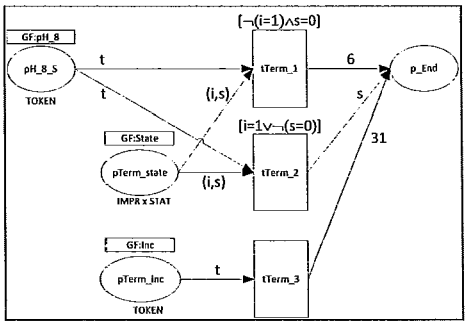

Figure 25: HCPN module TERMINATE instance; for ACME Corp.

The HCPN module TERMINATE waits for one of two events: either a TOKEN arrives in place pH_<n>_S or in place pTerm_inc. When the former occurs, this indicates that the execution of the last executed handler, handler_<x>, where $0 \leq x \leq n-1$, did not result in the pausing of the PAM Stack execution. The latter indicates that the PAM Stack execution was paused. The functionality obtained from transitions tTerm1 and tTerm2, along with the place pTerm_state, executes the termination of the PAM Stack execution, as the last step before the PAM Stack execution ends, as specified in step 5 in the algorithm in Table 13 on page 29 . The HCPN module TERMINATE implements the functionality specified via the state machine shown in Figure 8 on page 31.

Example: "Combining" DISPATCH with INTIALIZE and TERMINATE

As an example of how HCPN instances are combined to create a single, "partially" unfolded HCPN instance, we show a partial unfolding of the HCPN module DISPATCH. For the purpose of illustration, this particular partial unfolding only replaces the Initialize and Terminate substitution transitions with the INITIALIZE and TERMINATE HCPN modules, but does not replace the Handlers substitution transition with the HCPN module

HANDLERS. Also, in this partial unfolding example, we do not merge all fusion sets. For some fusion sets we do partial merging. For example, for the GF:ph_0 fusion set, we merge the place pH_Start_to_0 from the HCPN module INITIALIZE instance, with the place pH_0_S from the HCPN module HANDLERS instance. This, in effect, joins the HCPN module INITIALIZE and HANDLERS instances. Similarly, for the fusion set GF:ph_4, place pH_4_S from the HCPN module HANDLERS instance, and the place pH_4_S (same name, different HCPN Module) from the HCPN module TERMINATE are also merged, thus, joining the HCPN module HANDLERS and TERMINATE instances. The resulting partially unfolded HCPN is shown in Figure 26.

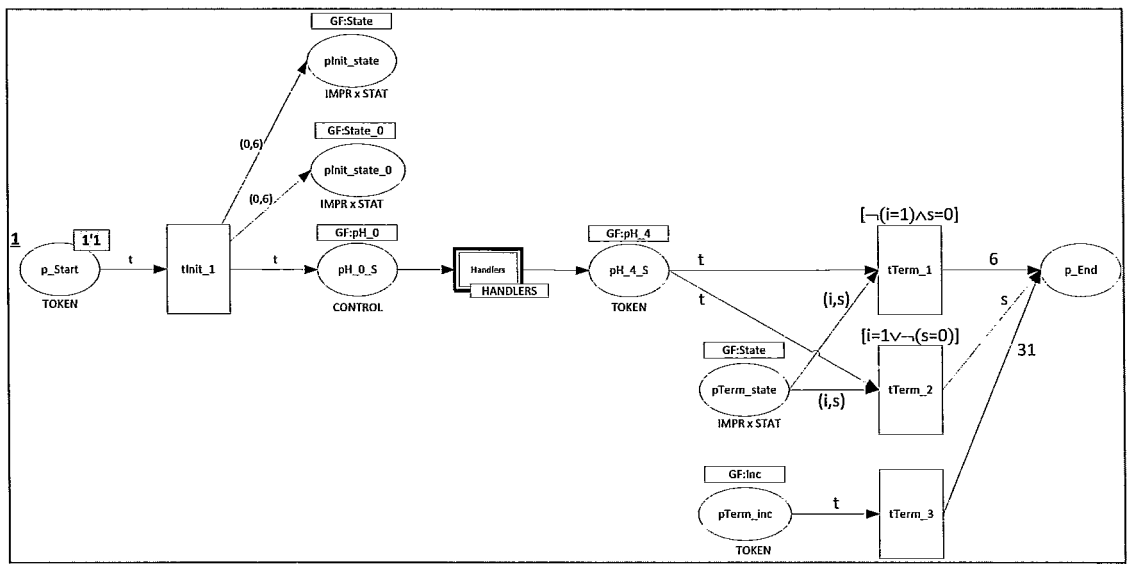

Figure 26: Partial unfolding of the HCPN module instance DISPATCH "combining" DISPATCH with INITIALIZE and TERMINATE HCPN module instances; for ACME Corp.

HCPN module HANDLERS

Given a PAM Stack Instance

${}^{T}\Pi_S^f = \{\text{handler}_i\} = \text{handler}_0, \text{handler}_1, \ldots, \text{handler}_{n-1}$ containing n handlers, the HANDLERS HCPN module models the execution of all possible handler subsequence executions allowable by the PAM Stack Instance. The pamtester-fm template for the HCPN module HANDLERS, shown in Figure 27, is used to generate the HCPN module HANDLERS specification for ${}^{T}\Pi_S^f$.

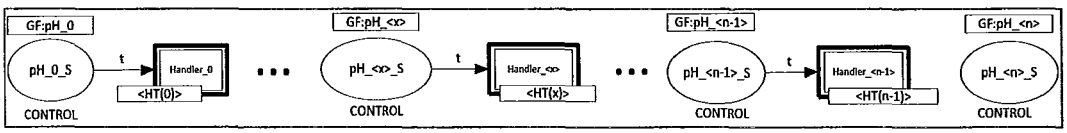

Figure 27: Pamtester-fm HCPN module HANDLERS template

The number n denotes the number of handlers comprising the PAM Stack Instance. The number x denotes an arbitrary number from 0 to $n-1$, i.e. $0 \leq x \leq n-1$. HT is a function, $HT: \{0, 1, \ldots, n-1\} \rightarrow \{NOT\_SUBSTACK\} \cup \{SUBSTACK\_0, SUBSTACK\_1, \ldots, SUBSTACK\_15\}$. The elements of the range of HT, i.e. $Ran(HT)$, are HCPN modules.

Pamtester-fm uses this HCPN module HANDLERS template to generate the corresponding HCPN module HANDLERS specification. The numbers n and the function HT are computed at runtime, during this generation.

The number n is obtained by pamtester-fm by parsing the data structure containing the PAM Stack Instance. Precisely, the number n equals the number of struct handler data structures comprising the PAM Stack Instance.

Also, as previously mentioned, each handler data structure contains information, including: handler type, substack level, name of "stacked" PAM corresponding to this handler, and last, but not least, a pointer to the "stacked" PAM's implementation of the Management Function corresponding to the PAM Stack Instance. Given a handler x, $0 \leq x \leq n-1$, Pamtester-fm extracts this information from the x^{th} handler data structure, thereby calculating the value of $HT(x)$.

The function HT identifies which HCPN module to associate with the Handler_<x> substitution transition, $0 \leq x \leq n-1$. HT classifies handlers into two categories: the handler is a start of a substack, i.e. type of handler is PAM_HT_SUBSTACK; the handler is of type PAM_HT_MODULE or PAM_HT_MUST_FAIL. In the former case, HT identifies which substack level, the substack level being between 0 and 15, this handler is associated with. Given a handler _x whose substack level is L, then HT(x)=SUBSTACK_L.

The reason why we chose to make this distinction in our HCPN modeling is because handlers that are not of type PAM_HT_SUBSTACK have the same execution structure. Specifically, handlers whose type is PAM_HT_MODULE or PAM_HT_MUST_FAIL share the same execution behaviour. This behaviour consists of, first, returning a PAM_RETURN value, and then executing an Action as dictated by the corresponding Control function. This can be seen in the PAM Stack Execution algorithm shown in Table 13 on page 29.

Figure 28 shows the HCPN module HANDLERS instance for Acme Corp. In this instance, all handler instances are not of type PAM_HT_SUBSTACK, i.e. the corresponding PAM Stack Instance does not contain any substacks.

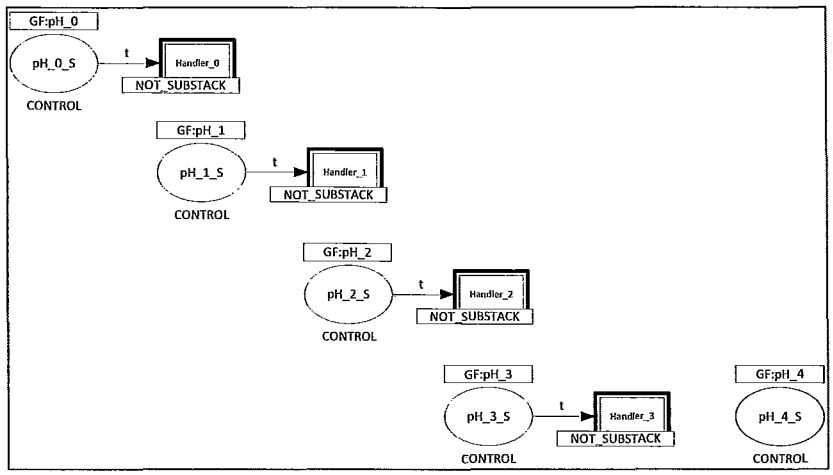

Figure 28: HCPN module HANDLERS instance; for ACME Corp.

Given a place pH_<x>_S, $0 \leq x \leq 3$, when a CONTROL token is placed in pH_x_S, this represents the act of choosing the x^{th} handler as the next handler to be executed. Precisely, this corresponds to steps 3.a and 4.h.i in the PAM Stack Execution algorithm (Table 13 on page 29), depending if $x = 0$, or $1 \leq x \leq 3$, respectively.

As discussed above, when a CONTROL token is placed in pH_4_S, this signifies the end of processing of handlers, and that the next step is to execute the termination portion of the PAM Stack execution (step 5 of algorithm in Table 13 on page 29).

As done previously, given the HCPN module HANDLERS instance example in Figure 28, we can continue our "partial" unfolding from Figure 26 on page 55. Figure 29 shows this continuation with the Handlers substitution transition being replaced by the HCPN module HANDLERS instance from Figure 28.

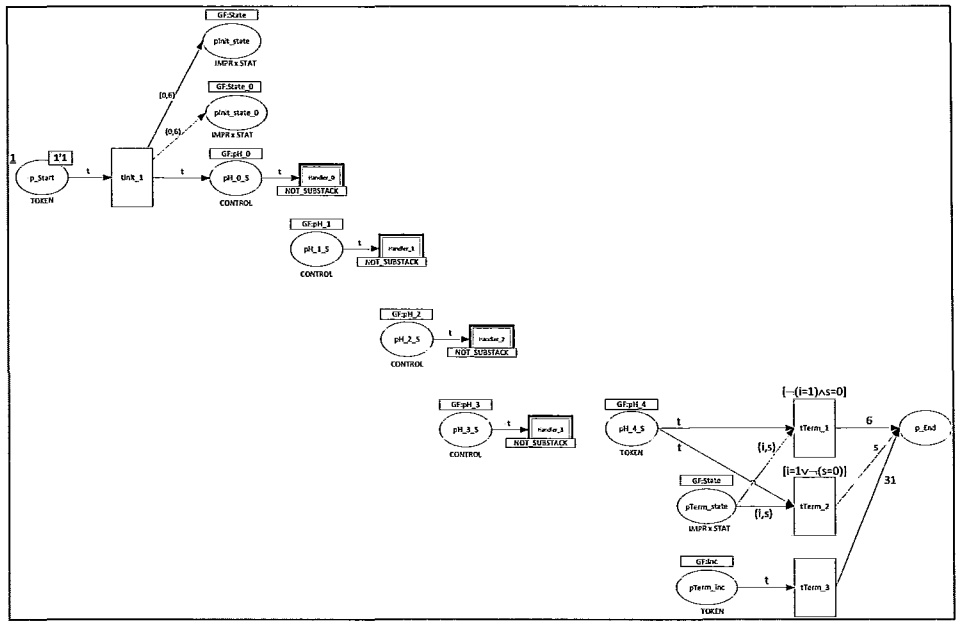

Figure 29: Partial unfolding of the HCPN module instances DISPATCH, INITIALIZE, HANDLERS, TERMINATE; for ACME Corp

HCPN module SUBSTACK

The pamtester-fm HCPN module SUBSTACK template is shown Figure 30.

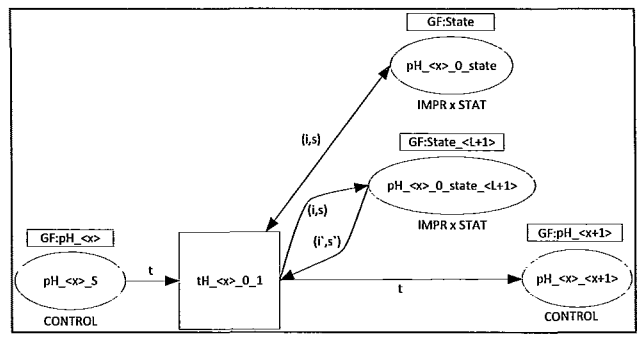

Figure 30: Pamtester-fm HCPN module SUBSTACK template

A placement of a CONTROL token in place pH_<x>_S enables the transition tH_<x>_0_1. Firing of this transition causes the storing of the current PAM Stack

Execution State (i,s) in the fusion set GF:State_<L+1>, where the current substack level of handler_<x> is L. The current PAM Stack Execution State (i,s) is obtained from the place pH_<x>_0_state, a member of the GF:State fusion set. When (i,s) is obtained, it is also replaced, so that subsequent transitions needing to obtain the current PAM Stack Execution state will be able to do so. The last Substack Level L+1 Execution State (i', s') is effectively deleted as the transition consumes it and does not generate any copies of it. Additionally, a CONTROL token is placed in place pH_<x>_to_<x+1>, which belongs to the GF:pH_<x+1> fusion set. This represents the choosing of the $(x+1)^{st}$ handler (the next handler in the sequence) as the handler to be executed next.

This HCPN module SUBSTACK template specification implements the "entering of a new substack level", as outlined in steps 4.a.i (saving of current PAM Stack Execution State) and 4.c.i (choosing next handler to be executed as the next handler in the sequence) of the PAM Stack Execution algorithm shown in Table 13 on page 29.

HCPN module NOT_SUBSTACK

The handlers of type PAM_HT_MODULE and PAM_HT_MUST_FAIL use the HCPN module NOT_SUBSTACK template, as shown in Figure 31.

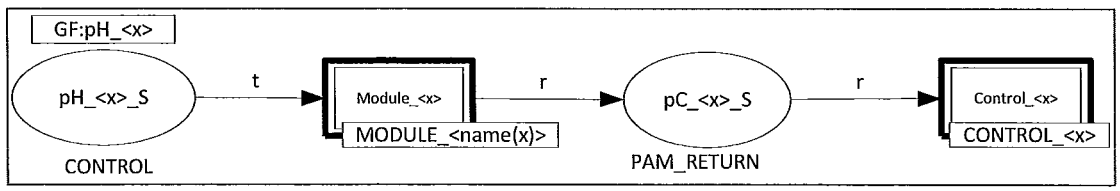

Figure 31: Pamtester-fm HCPN module NOT_SUBSTACK template

Execution of handler types PAM_HT_MODULE, and PAM _HT_MUST_FAIL have the same execution structure. First, a PAM_RETURN value is obtained. This is represented by the substitution transition Module_<x>. Second, the obtained PAM_RETURN value is used to execute the appropriate Action, this Action being determined via the corresponding Control function. This is represented by the substitution transition Control_<x>. Here, the obtained PAM_RETURN value not only controls PAM Stack Execution, i.e. a PAM_RETURN token is placed in the place pC_<x>_S, but the PAM_RETURN token also carries information to the next HCPN module as input, i.e. value of the PAM_RETURN token is used to compute the image of the corresponding Control function (this computation represented by Control_<x>), or in other words, this value is used to determine the Action to execute.

Precisely, as shown in the PAM Stack Execution algorithm (Table 13 on page 29), PAM Stack Execution first checks (step 4.b) if the handler is erroneous (handler is of type PAM_HT_MUST_FAIL). If so, then the error PAM_MUST_FAIL_CODE = 6 is returned. In this case, even though no PAM execution takes place, we can interpret the action of returning this error code as a "PAM or module execution". In this case, name(x):=MUST_FAIL, and the value of the substitution tag becomes MODULE_<name(x)>:=MODULE_MUST_FAIL.

If handler is not of type PAM_HT_MUST_FAIL, then we have two sub-cases:

- the corresponding PAM P_x contains an implementation $I(f^{sm}, P_x)$ of the Module API function f^{sm} corresponding to the Management Function f associated with the PAM Stack Instance ${}^{T}\Pi_S^f$ in question, or

- otherwise, P_x does not contain $I(f^{sm}, P_x)$.

Thus, the next relevant check carried out by the PAM Stack Execution algorithm (step 4.d) is to check whether P_x does not implement f^{sm}, i.e P_x does not contain $I(f^{sm}, P_x)$. If true, then the error PAM_MODULE_UNKNOWN = 28 is returned. Again, although no PAM execution takes place, we interpret the action of returning of the error code as a "PAM or module execution". In this case, name(x):=FUNC_NULL, and the value of the substitution tag becomes MODULE_<name(x)>:=MODULE_FUNC_NULL.

Lastly, if P_x does implement f^{sm}, i.e P_x contains $I(f^{sm}, P_x)$, then PAM Stack executes $I(f^{sm}, P_x)$, supplying the PAM options O_x as arguments, and obtains a PAM_RETURN code $return_x$, i.e. $\text{return}_x := \text{exec}(I(f^{sm}, P_x), O_x)$. In this case, name(x):=s ∈ PAMS = { SECURETTY, ENV, UNIX, SUCCEED_IF, ...}, where s corresponds to the module P_x, and the substitution tag becomes MODULE_<name(x)>:=MODULE_<s>. For example, if handler_x corresponds to pam_unix.so PAM, and handler_x is not erroneous, and pam_unix.so PAM implements management function f^{sm}, then name(x) = s = UNIX is a member of PAMS and MODULE_<name(x)> := MODULE_<s> = MODULE_UNIX.

To summarize, in all three of the above cases (i.e. check for an erroneous handler via step 4.b, check for existence of appropriate function implementation via step 4.d, execution of function implementation via step 4.e) a PAM_RETURN code is obtained (i.e. $\text{return}_x := 6, \text{return}_x := 28, \text{return}_x := \text{exec}(I(f^{sm}, P_x), O_x)$, respectively). The modeling of the returning of this PAM_RETURN code is done by the HCPN module associated with the Module_<x> substitution transition. Once the return value is obtained, the appropriate Action is determined (action 4.g) via the corresponding Control function,

i.e. $\text{action}_x := C_x(\text{return}_x)$. The modeling of determining the appropriate Action is done by the HCPN module associated with the Control_<x> substitution transition.

Continuing with our partial unfolding example for ACME Corp (Figure 29 on page 59), assuming that all 4 handlers are not erroneous, and that all 4 handlers implement the pam_sm_authenticate() Management Function, we obtain the unfolding in Figure 32. This unfolding is obtained by replacing all Handler_<x> substitution transitions with HCPN module NOT_SUBSTACK instances.

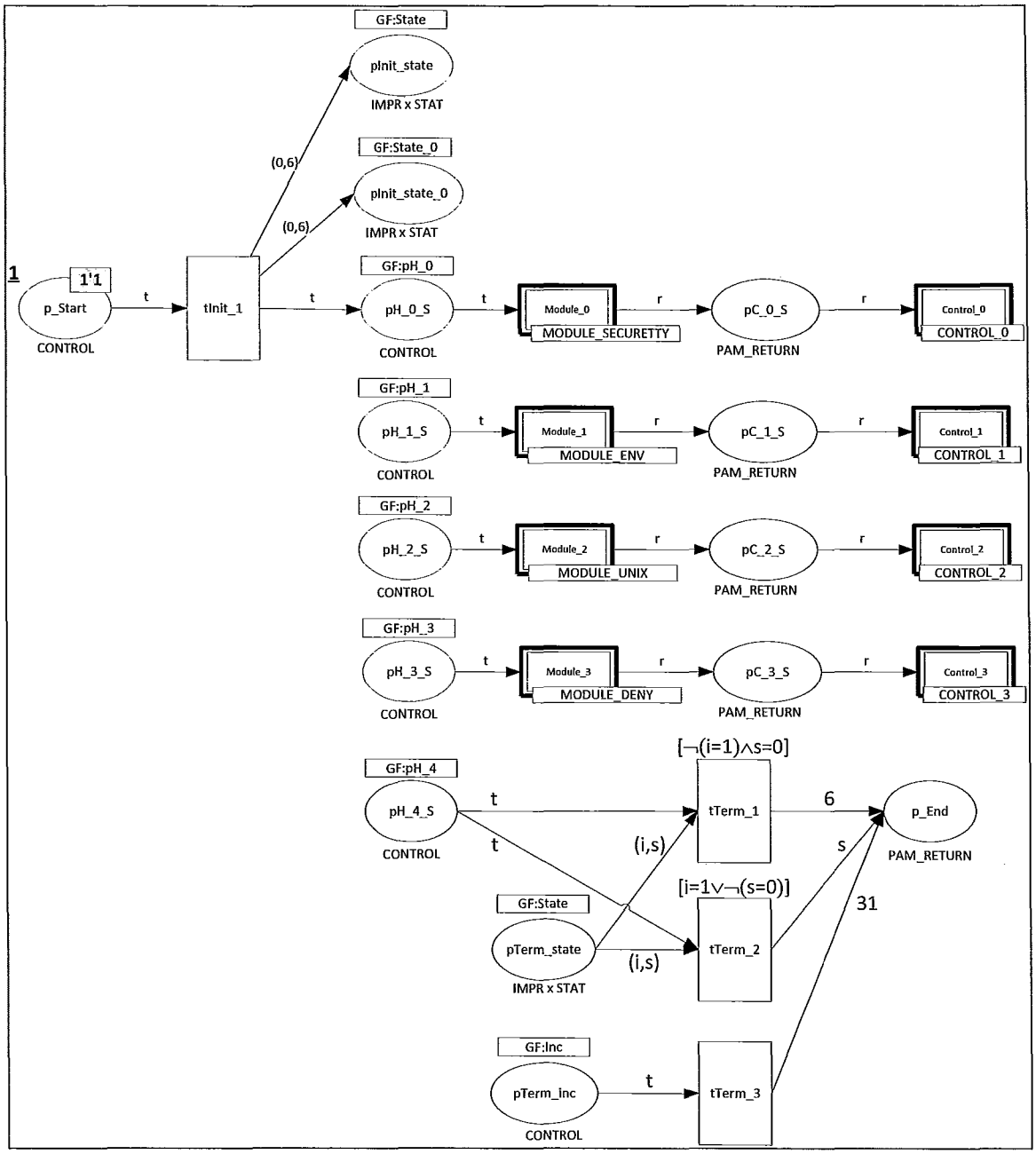

Figure 32: Partial unfolding of the HCPN module instances DISPATCH, INITIALIZE, HANDLERS, NOT_SUBSTACK, TERMINATE; for ACME Corp

HCPN module MODULE_<name(x)>

As previously discussed, the HCPN module MODULE_<name(x)> has three cases:

- the handler is erroneous, i.e. HCPN module MODULE_MUST_FAIL;
- the management function is not implemented, i.e. HCPN module MODULE_FUNC_NULL;
- the handler is not erroneous and the handler implements the appropriate management function, i.e. HCPN module MODULE_<s>, where string s identifies a PAM, string s is a member of PAMS={SECURETTY, ENV, UNIX, SUCCEED_IF,...}.

HCPN module MODULE_MUST_FAIL

The pamtester-fm HCPN module MODULE_MUST_FAIL template is shown in Figure 33.

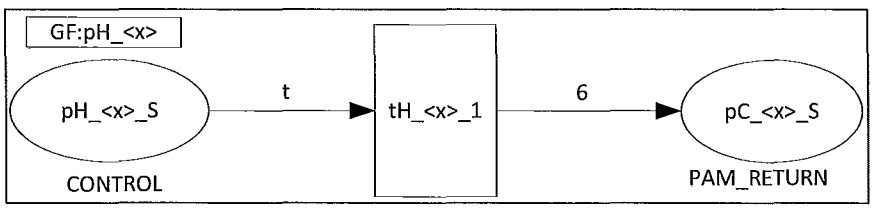

Figure 33: Pamtester-fm HCPN module MODULE_MUST_FAIL template

All HCPN module MODULE_MUST_FAIL instances simply return a PAM_RETURN error code PAM_MUST_FAIL_CODE = 6.

HCPN module MODULE_FUNC_NULL

The pamtester-fm HCPN module MODULE_FUNC_NULL template is shown in Figure 34.

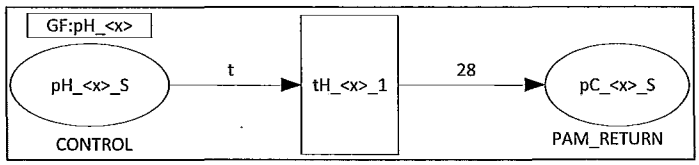

Figure 34: Pamtester-fm HCPN module MODULE_FUNC_NULL template

All HCPN module MODULE_FUNC_NULL instances simply return a PAM_RETURN error code PAM_MODULE_UNKNOWN = 28.

HCPN module MODULE_<s>

Given a PAM P_x "stacked" on a PAM Stack instance ${}^{T}\Pi_{S}^{f}$, if P_x implements the corresponding Module API function f^{sm}, i.e. $I(f^{sm}, P_x)$, then pamtester-fm uses the HCPN module MODULE_<s> template to generate the appropriate HCPN module instance which models the execution of $I(f^{sm}, P_x)$. The pamtester-fm HCPN module MODULE_<s> template is as shown in Figure 35.

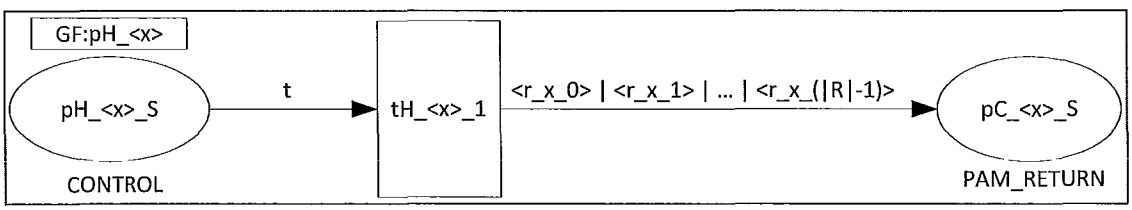

Figure 35: Pamtester-fm HCPN module MODULE_<s> template

In our current model, the model of an "execution" of a PAM P_x, HCPN module MODULE_<s>, consists of a single transition tH_<x>_1. This transition is defined to consume a single CONTROL token. Also, this transition generates a single PAM_RETURN token. The value of this PAM_RETURN token is taken from R, the set of possible values that can be returned by the execution of $I(f^{sm}, P_x)$. This is modeled by the transition tH_<x>_1 being defined as capable of returning, and only returning, exactly one of the values <r_x_0>, <r_x_1>, …, <r_x_(|R|-1)>, where $|R|$ is the cardinality of the set R, and $\forall\, k \in 0..|R|-1$, r_x_k $\in R$. We denote the fact that only one of the values can be returned by the firing of the transition tH_<x>_1 by using the '|' character to separate the return values, i.e. <r_x_0> | <r_x_1> | … | <r_x_(|R|-1)>.

The HCPN module MODULE_<s> template is then used to generate the templates for each of the PAMs, i.e. MODULE_SECURETTY, MODULE_ENV, MODULE_UNIX, etc.

For example, Figure 36 shows a pamtester-fm template for the pam_securetty.so PAM, i.e. HCPN module MODULE_SECURETTY. In our modeling, we assume that the pam_securetty.so PAM is capable of returning only the following PAM_RETURN values: PAM_SUCCESS = 0, PAM_SERVICE_ERR = 3, PAM_AUTH_ERR = 7, PAM_IGNORE = 25 and PAM_INCOMPLETE = 31. In this case, $|R| = 5$, and the possible return values are obtained using the following derivation process: <r_x_0> = r_<x>_0 := 0, <r_x_1> = r_<x>_1 := 3, <r_x_2> =

r_<x>_2:= 7, <r_x_3> = r_<x>_3 := 25 and <r_x_4> = r_<x>_4 := 31.

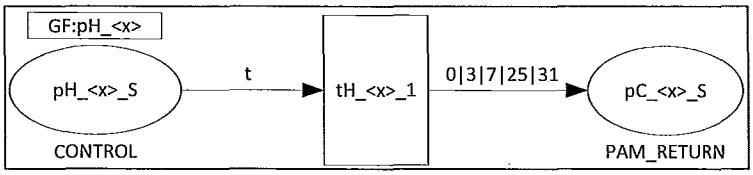

Figure 36: Pamtester-fm HCPN module MODULE_SECURETTY template

For example, consider the HCPN module MODULE_SECURETTY template in Figure 36. Furthermore, recall that the PAM Stack instance for ACME Corp. (Figure 94 on page 185), contains the pam_securetty.so PAM as the first "stacked" PAM, i.e. handler_0 is associated with pam_securetty.so PAM. Then, the HCPN module MODULE_SECURETTY instance is as shown in Figure 37. In this case, <r_x_0> = r_0_0 := 0, <r_x_1> = r_0_1 := 3, <r_x_2> = r_0_2:= 7, <r_x_3> = r_0_3 := 25 and <r_x_4> = r_0_4 := 31.

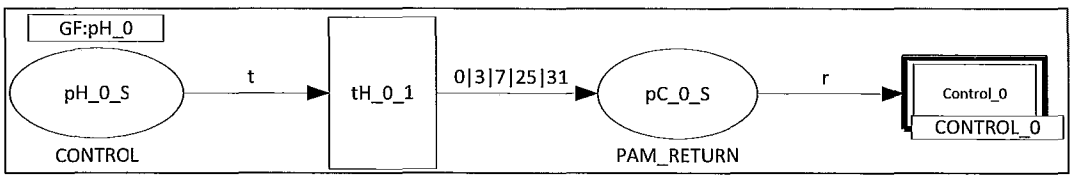

Figure 37: HCPN module MODULE_SECURETTY instance - ACME Corp

In this case, continuing with the partial unfolding example for ACME Corp (Figure 32 on page 64), we obtain the unfolding in Figure 38. This is done by replacing the substitution transition Module_0 with an HCPN module MODULE_SECURETTY instance shown in Figure 37.

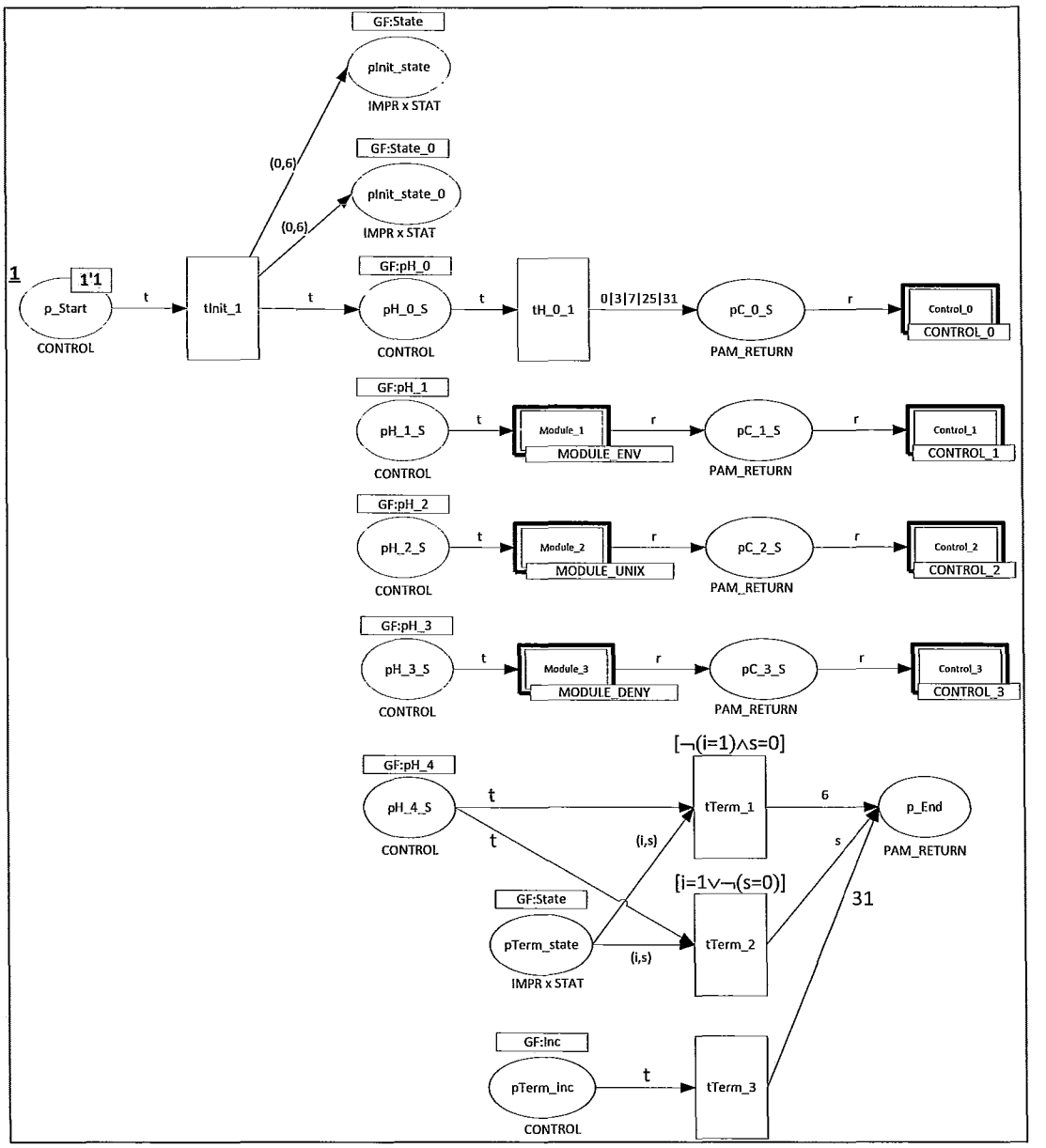

Figure 38: Partial unfolding of the HCPN module instances DISPATCH, INITIALIZE, HANDLERS, NOT_SUBSTACK, MODULE_SECURETTY, TERMINATE; for ACME Corp

HCPN module CONTROL

The pamtester-fm HCPN module CONTROL template is shown in Figure 39.

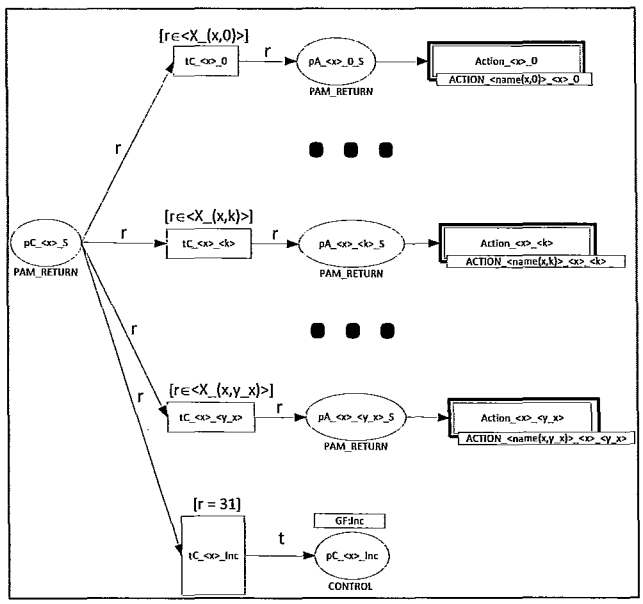

Figure 39: Pamtester-fm HCPN module CONTROL template

This template generates the HCPN module CONTROL instances for each handler whose type is not PAM_HT_SUBSTACK. This template implements the PAM Stack execution functionality of obtaining a PAM_RETURN value, and determining which Action to execute based on this PAM_RETURN value. This functionality corresponds to step 4.g in the PAM Stack Execution algorithm (Table 13 on page 29). HCPN module CONTROL also handles the "pausing" of PAM Stack Execution (step 4.f in Table 13 on page 29).

Given handler_x, such that type of handler_x is not PAM_HT_SUBSTACK, the determination of which Action to execute is defined by the corresponding Control function C_x.

The value of the PAM_RETURN token in place pC_<x>_S corresponds to the value of return_x in step 4.g in the PAM Stack Execution algorithm (Table 13 on page 29). This value is an integer in the range [0,31].

The place pC_<x>_S belongs to the preset of exactly the following set of transitions: tC_<x>_0, …, tC_<x>_<k>, …, tC_<x>_<y_x>, tC_<x>_Inc. For the k^{th} transition, $0 \leq k \leq y\_x$, the transition guard evaluates to TRUE if, and only if, the PAM_RETURN value r belongs to the guard's range of PAM_RETURN values X_(x,k), i.e. r ∈ X_(x,k).

For the transition tC_<x>_Inc, we define this range by the the singleton set {31}, and we denote it by the number 31. In this case, we set the guard to be [r = 31]. For transition tC_<x>_<k>, <X_(x,k)> is the range template, and [r ∈ <X_(x,k)>] is the guard template.

The membership of the set denoted by X_(x,k) is obtained from the definition of the Control function, C_x (contained in the corresponding handler_x) using the algorithm described in Table 12 on page 27, i.e. X_(x,k) := k^{th} partition X in partition sequence $\{X\}$, denoting a range of PAM_RETURN values, each of these return values having the same image under corresponding Control function, i.e. $\forall r \in \text{X\_(x, k)}: C_x(r) = a, a$ constant. Thus, given the set of partitions $\{X\} \coloneqq \{\text{X\_(x, 0)}, \text{X\_(x, 1)}, \ldots, \text{X\_(x, k)}, \ldots, \text{X\_(x, y\_x)}\}$ (see algorithm in Table 12 on page 27 to see how such a set of partitions is obtained), we use each partition $\text{X\_(x, k)}, k \in 0, 1, \ldots, y\_x$, as the range for the corresponding transition guard of the transition tC_<x>_<k>, i.e. [r ∈ <X_(x,k)>]. The symbol used to represent the last guard [r ∈ <X_(x,y_x)>] is "ow" (shorthand, since this range contains the most return values). As Linux-PAM configurations tend to have one large partition and a few small ones, and given that the set of PAM_RETURN values contains 31 elements, hence,

for display purposes, we chose to use the symbol "ow" for the renderings of the guard function for the most populous partition (with largest minimal value).

As explained above, the set of PAM_RETURN values is partitioned among the guards of the transitions containing pC_<x>_S in their preset. Hence, putting a PAM_RETURN token in place pC_<x>_S causes exactly one of these transitions to become enabled. We have two cases for this enabled transition: a single transition tC_<x>_k is enabled, for some k in {0,1, .., <y_x>}; or the transition tC_<x>_Inc is enabled. Continuing with the example from Figure 37 on page 68, Figure 40 shows the substitution transition Control_0 replaced with its corresponding HCPN module CONTROL_0 instance. Here, $\{X\} := \{X\_(0,0), X\_(0,1), X\_(0,2)\}$, $X\_(0,0) := \{0,12\}$, $X\_(0,1) := \{25\}$ and $X\_(0,2) := \{1,2,3,4,5,6,7,8,9,10,11,13,14,15,16,17,18,19,20,21,22,23,24,26,27,28,29,30\}$.

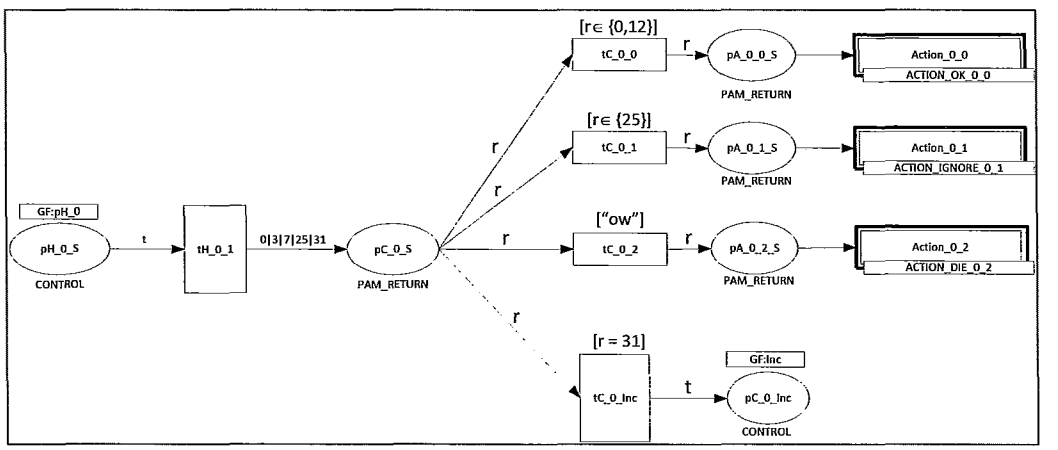

Figure 40: HCPN module CONTROL_0 instance - ACME Corp

Pausing of PAM Stack Execution

If tC_<x>_Inc is enabled, this implies that the value of the PAM_RETURN token in pC_<x>_Inc is 31. This situation corresponds to step 4.f in the PAM Stack Execution algorithm (Table 13 on page 29).

If PAM Stack execution is paused, then no Action is executed. Instead, PAM Stack execution is stopped, and execution control is returned to the Linux-PAM Client.

This "pausing" is implemented as follows. When tC_<x>_Inc is fired, it consumes the PAM_RETURN token from pC_<x>_S and deposits a CONTROL token in in the place pC_<x>_Inc. This corresponds to step 4.f in the PAM Stack Execution algorithm (Table 13 on page 29). At this point, the next enabled transition is tTerm_3. The transition tTerm_3 belongs to the HCPN module TERMINATE. Thus, effectively, enabling of this transition chooses the termination portion of PAM Stack execution. This corresponds to step 6 in the PAM Stack Execution algorithm (Table 13 on page 29).

For instance, the pam_sm_authenticate() Module API function implementation of pam_securetty.so PAM is capable of returning PAM_RETURN value of PAM_INCOMPLETE = 31. A return value of 31 indicates that the pam_securetty.so PAM deems that it has not received sufficient information to carry out user authentication-related functionality.

For example, the ACME Corp's PAM Stack instance for the pam_authenticate() Management Function, the 1st handler, handler_0, is associated with the pam_securetty.so PAM. The following figures illustrate the "pausing" of PAM Stack execution. First, handler_0 is chosen to be executed by placing a single CONTROL token of value 1 in pH_0_S (signified by the grey rectangle containing the text 1`1). At this point, transition tH_0_1 is enabled (signified by grey background of tH_0_1). This is shown in Figure 41. Note that place pC_0_Inc was merged with place pTerm_inc, since both places belong to the same fusion set, namely GF:Inc. Here, WLOG, we show place pTerm_inc.

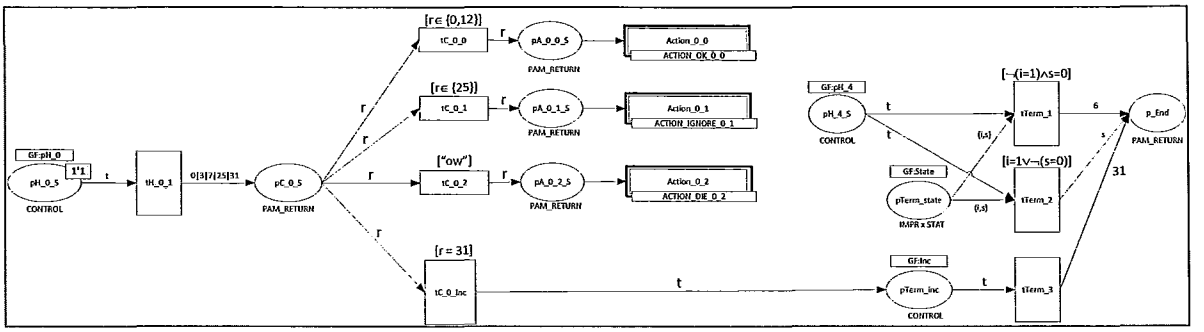

Figure 41: "Pausing" of PAM Stack – handler_0 is chosen to be executed

Suppose that pam_securetty.so PAM returns PAM_INCOMPLETE return code (Figure 42), i.e. transition tH_0_1 consumes the CONTROL token from place pH_0_S, and deposits a PAM_RETURN token of value 31 into place pC_0_S. Then, transition tC_0_Inc becomes enabled, since its guard, [r = 31] evaluates to TRUE (since the value of the token in place pC_0_S is 31).

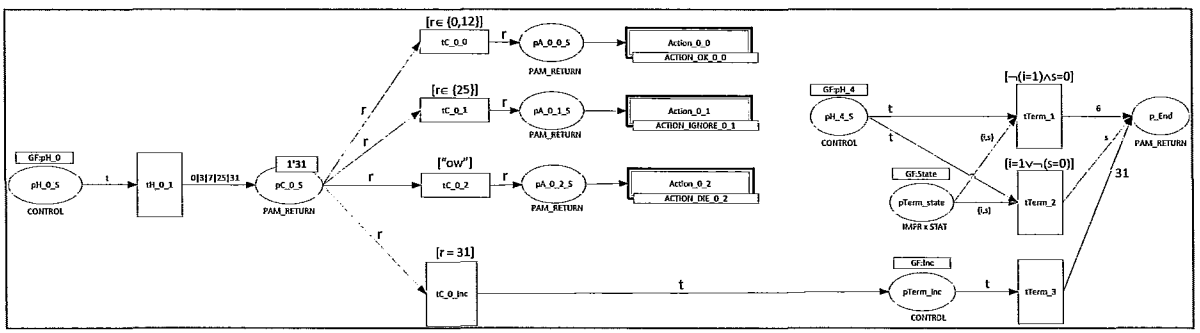

Figure 42: "Pausing" of PAM Stack – pam_securetty.so PAM returns PAM_INCOMPLETE = 31

When transition tC_0_Inc fires, it consumes the token 1'31 from place pC_0_S and deposits a CONTROL token 1'1 in place pTerm_Inc. Hence, transition tTerm3 is enabled (Figure 43).

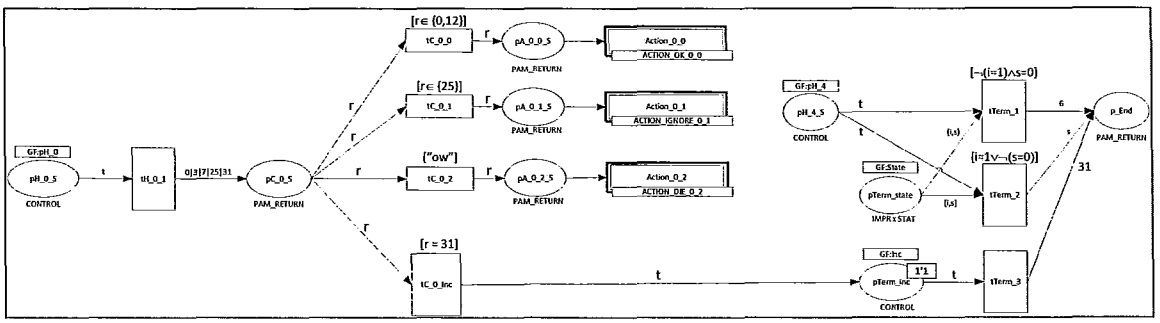

Figure 43: "Pausing" of PAM Stack – HCPN module TERMINATE is chosen to be executed

Lastly, once transition tTerm_3 fires, then tTerm3 consumes the CONTROL token 1'1 from pTerm_Inc, and places a single PAM_RETURN token of value 31 in the place pEnd, 1'31, effectively terminating the PAM Stack execution (Figure 44).

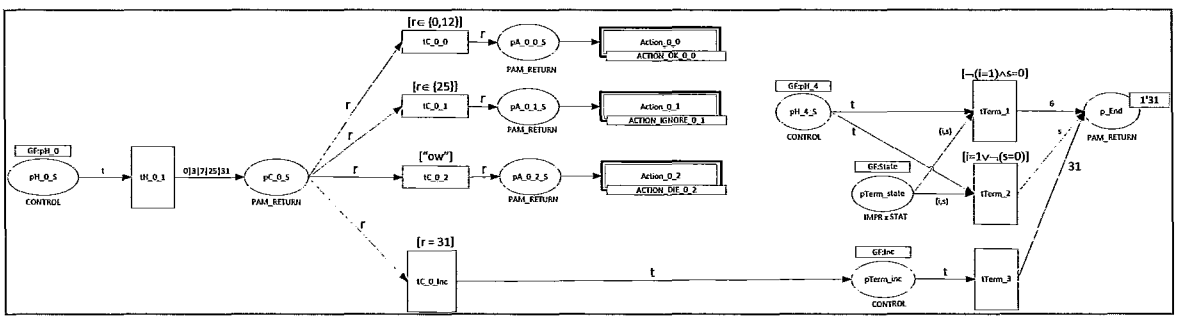

Figure 44: "Pausing" of PAM Stack – PAM Stack execution terminates

Choosing to Execute an Action

If transition tC_<x>_<k>, <k> ∈ 0..<y_x>, is enabled, then the firing of this transition effectively chooses an Action to be executed. This is implemented by tC_<x>_<k> removing the PAM_RETURN token from the place pC_<x>_S, and placing this same PAM_RETURN token in the starting place of the Action to be executed, i.e. in place pA_<x>_<k>_S, <k> ∈ 0..<y_x>. This removal/placement of the PAM_RETURN token

not only dictates PAM Stack Execution control, but also passes data (the PAM_RETURN value) to the Action to be executed. Note that there are as many distinct Actions as there are partitions $\{X\} := \{X\_(x,0), X\_(x,1), \ldots, X\_(x,k), \ldots, X\_(x,y\_x)\}$. Each partition corresponds to a distinct Action.

For example, continuing with the partial HCPN model instance shown in Figure 41 on page 74, suppose that the pam_securetty.so PAM returns the PAM_RETURN = PAM_SUCCESS = 0. Then, a PAM_RETURN token of value 0, 0'0, is placed in pC_0_S. This causes transition tC_0_0 to become enabled. This is because value of token in pC_0_S is 0, and 0 belongs to the tC_0_0 transition guard's range of values. Specifically, if $0 \in X\_(0,0) = \{0,12\}$, then $[r \in \{0,12\}]$ evaluates to TRUE (Figure 45).

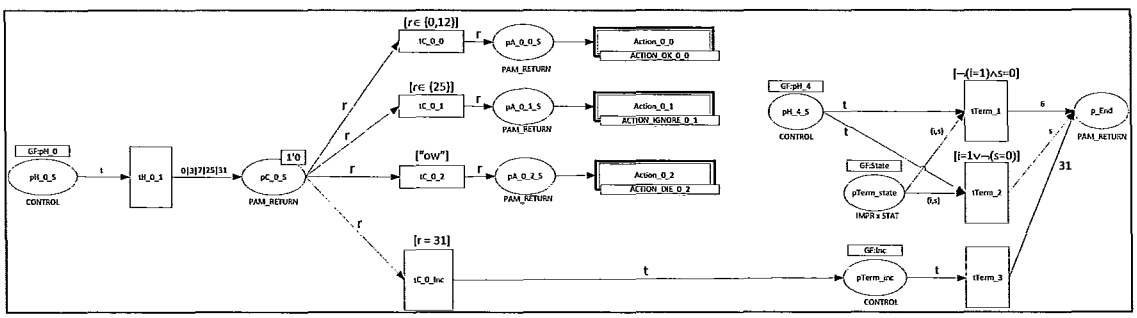

Figure 45: Choosing an Action to execute based on PAM_RETURN value defined by Control – pam_securetty.so PAM returns PAM_SUCCESS = 0

Then, once transition tC_0_0 fires, the PAM_RETURN token 1`0 in place pC_0_S is consumed by transition tC_0_0, and a PAM_RETURN token 1`0 is placed in pA_0_0_S. Hence, the Action associated with the substitution transition Action_0_0 (HCPN module ACTION_OK_0_0 instance) is chosen for execution (Figure 46).

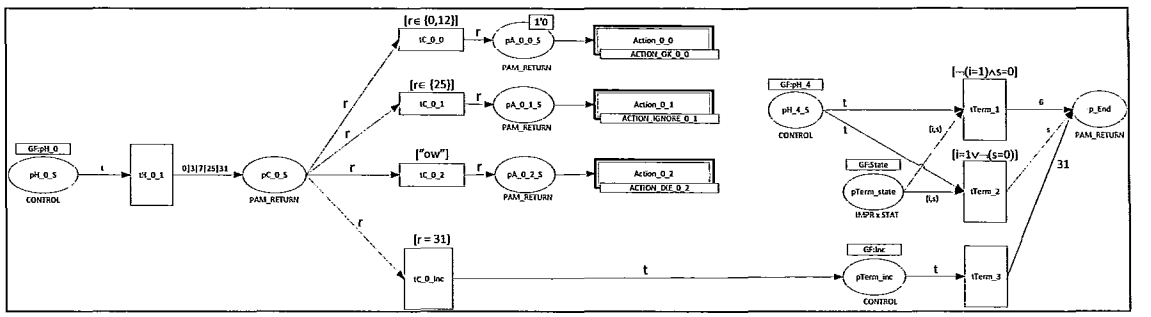

Figure 46: Choosing an Action to execute based on PAM_RETURN value defined by Control – Action associated with Action_0_0 is chosen for execution

During the generation of HCPN module CONTROL instances, pamtester-fm determines the names of the corresponding actions. Given the k^{th} Action substitution transition for the x^{th} handler, Action_<x>_<k>, the corresponding substitution tag has the form ACTION_<name(x,k)>_<x>_<k>. The symbol namc(x,k) denotes the result of a computation that determines the name of the Action to be used as the k^{th} action of the x^{th} Control.

For example, in Figure 46, given the first handler, i.e. x=0, and the first action, i.e. k=0, then name(x,k)=name(0,0)=OK, meaning that the first action of the first handler is Action OK. In general, since each HCPN module ACTION has a different HCPN specification, thus, pamtester-fm generates a different HCPN module for each Action. Hence, we suffix <x>_<k> to Action_<name(x,k)> to create the resulting substitution transition tag. Continuing our example, the first action of the first handler has a substitution tag ACTION_<name(x,k)>_<x>_<k>=ACTION_<name(0,0)>_0_0=ACTION_OK_0_0. In

this case, pamtester-fm will instantiate an HCPN module ACTION_OK_0_0 in order to substitute this module for the substitution transition Action_0_0.

HCPN module templates for Actions

There are 6 types of actions: Action 'ignore', Action 'ok', Action 'done', Action 'bad', Action 'die', and Action 'jump'.

All Actions have a similar HCPN structure. The first portion of each Action is devoted to updating of the PAM Stack Execution State. The second portion of each Action is devoted to choosing the next handler to be executed.

This structural and behavioural design decision was made to implement the functionality of the PAM Stack execution. Mainly, in the PAM Stack Execution algorithm, once an Action is executed, the execution of this Action does two things: update the PAM Stack Execution State, and choose the next handler to execute.

The PAM Stack Execution State is implemented by the fusion set GF:State. The colour of the places in this fusion set (the type of tokens accepted by these places) is a two-tuple (i,s), where i denotes Impression, and s denotes Status.

For pamtester-fm Action templates, <x>_<k_x> denotes k^{th} Action of x^{th} handler.

HCPN module ACTION_IGNORE

The HCPN module ACTION_IGNORE_<x>_<k_x> template is shown in Figure 47.

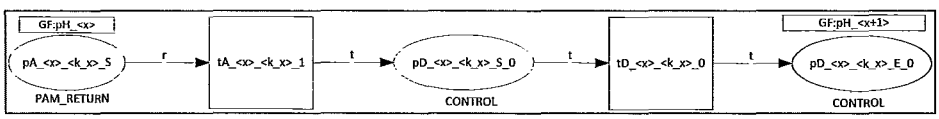

Figure 47: Pamtester-fm HCPN module ACTION_IGNORE template

Essentially, this Action "ignores" the received PAM_RETURN in a sense that this Action does not affect the state of the PAM Stack Execution. Specifically, the execution of this Action is as follows: tA_<x>_<k_x>_1 consumes the PAM_RETURN value from pA_<x>_<k_x>_S, and places a CONTROL token in pD_<x>_<k_x>_S_0 – this is the "ignore" portion. Then, this Action chooses the next handler in the sequence as the next handler to be executed. Specifically, the place pD_<x>_<k_x>_E_0 belongs to the fusion set defined by the substitution tag template GF:pH_<x+1>. Once a CONTROL token is placed in pD_<x>_<k_x>_E_0, this indicates that the handler whose starting place belongs to the fusion set identified by GF:pH_<x+1> is the handler to be executed next.

The substitution tag is generated by pamtester-fm during the generation of the HCPN. In this case, the "formula" for computing the next handler to be executed is x+1, where x is the current handler depth.

For example, given the 1st handler in ACME Corp's PAM Stack Instance, handler_0, if the pam_securetty.so PAM returns a PAM_RETURN value PAM_IGNORE = 25, then the ACTION_IGNORE_0_1 is chosen to be executed (see Figure 48). In this case, the next handler to be executed is the 2nd handler, handler_1, i.e. pD_0_1_E_0 belongs to the fusion set GF:pH_1.

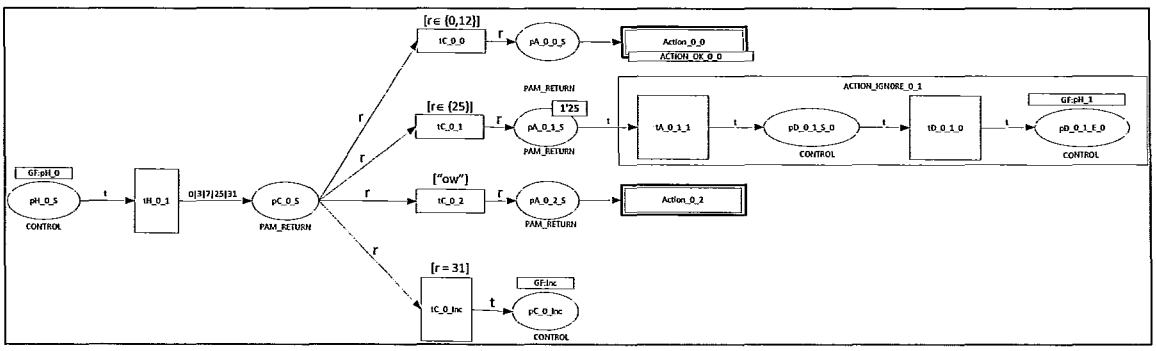

Figure 48: HCPN module ACTION_IGNORE_0_1 - ACME Corp

HCPM module ACTION_OK

The HCPN Template for Action 'ok' is shown in Figure 49.

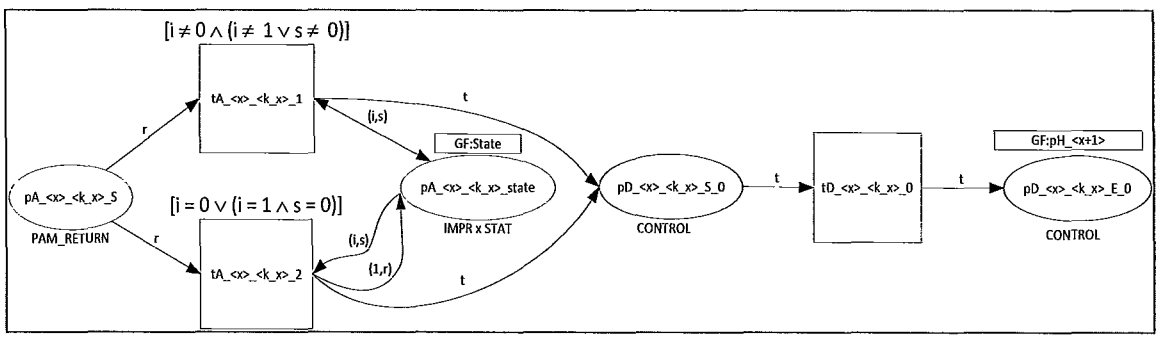

Figure 49: Pamtester-fm HCPN module ACTION_OK template

Depending on the current PAM Stack Execution State (i,s), the State may be updated or not. If it is, then it is updated to (1,r), where r is the PAM_RETURN just obtained.

The next handler to be chosen for execution is the next handler in the sequence.

HCPN module ACTION_DONE

The HCPN Template for Action 'done' is shown in Figure 50.

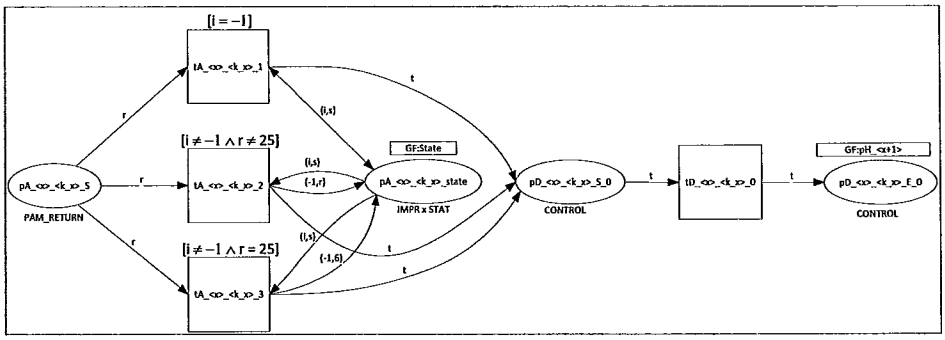

Figure 50: Pamtester-fm HCPN module ACTION_DONE template

PAM Stack Execution State update is the same as for Action 'ok'.

The choosing of the next handler depends on the current Impression of the PAM Stack Execution State. If the Impression is positive, i.e. PAM_POSITIVE = 1, then the next handler to be chosen for execution is the first subsequent handler, after the current depth x, whose substack level is less than the current substack level. This is computed by pamtester-fm using the skip_substack(x) function, for the current handler depth x. If the PAM Stack Instance only consists of a single substack level (this is the case in all production Linux-PAM Configurations tested so far), then skip_substack(x) "skips" all subsequent handlers, and PAM Stack Execution ends with the Terminate module.

On the other hand, if the Impression is not positive, then the next handler to be executed is the next handler in the sequence, i.e. the handler at depth x+1.

HCPN module ACTION_BAD

The HCPN Template for Action 'bad' is shown in Figure 51.

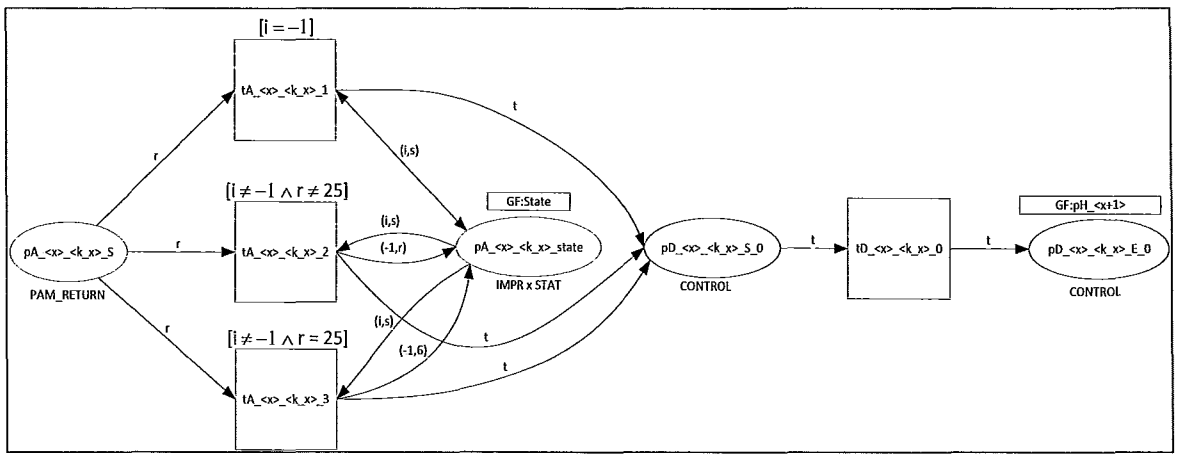

Figure 51: Pamtester-fm HCPN module ACTION_BAD template

Here, depending on the PAM Stack Execution State, and whether or not the current return value is 25, we may choose to update the PAM Stack Execution State. If the Impression is already negative, then we ignore current return value. Otherwise, we change the Impression to PAM_NEGATIVE = -1. We also update the Status. If the return value is PAM_IGNORE = 25, meaning that the PAM is essentially telling us (via this return value) to ignore its (the PAM's) execution result, then we update the Status with the default "bad" Status PAM_MUST_FAIL_CODE := PAM_PERM_DENIED = 6. Otherwise, if the return value is not PAM_IGNORE, i.e. $r \neq 25$, then we update the Status with this return value.

In all cases, the next handler to be executed is the next handler in the handler sequence.

HCPN module ACTION_DIE

The HCPN Template for Action 'die' is shown in Figure 52.

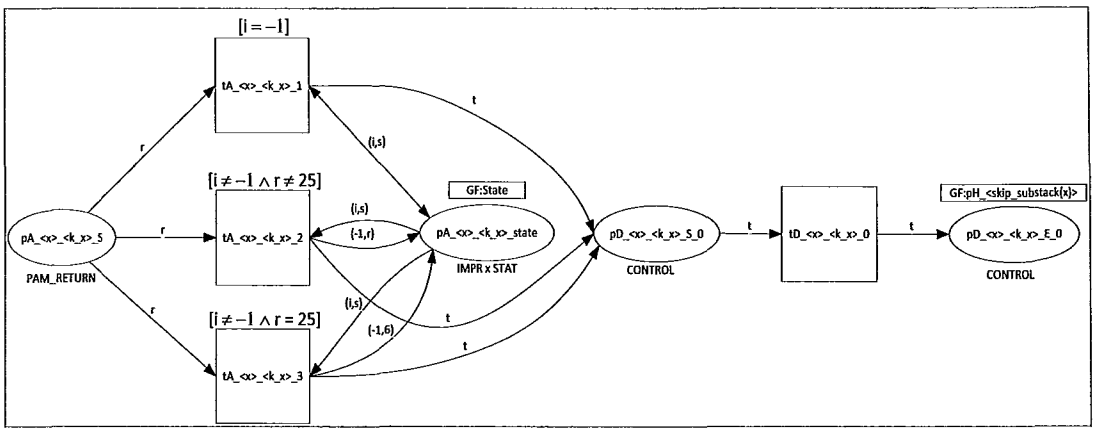

Figure 52: Pamtester-fm HCPN module ACTION_DIE template

PAM Stack Execution State update is the same as Action 'bad'.

The next handler to be executed uses the skip_substack(x) function, where x is the current handler depth. This is computed the same way as by Action 'done'.

One point worth noting here is that, interestingly, in contrast to the operation of Action 'done', the choosing of the next handler does the skip_substack(x) computation in all cases, whereas Action 'done' only "skipped substacks" if the Impression was PAM_POSITIVE = 1. In other words, Action 'die' skips the substack no matter what, whereas Action 'done' skips the substack conditionally.

HCPN module ACTION_RESET

The HCPN Template for Action 'reset' is shown in Figure 53.

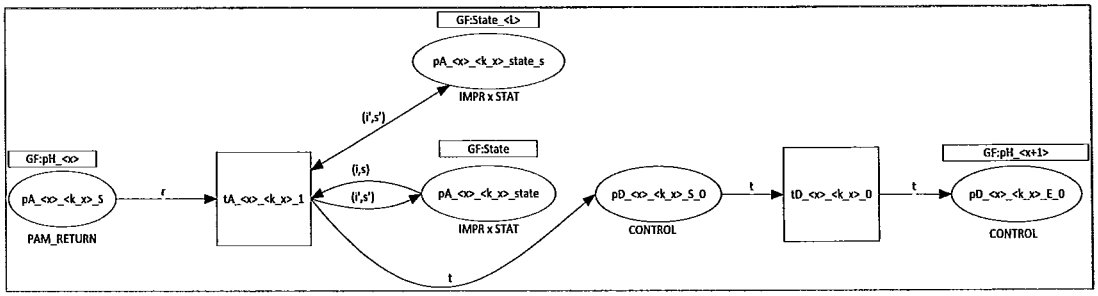

Figure 53: Pamtester-fm HCPN module ACTION_RESET template

The HCPN module RESET_<x>_<k_x> instance "resets" the Execution PAM Stack State to a previous PAM Stack Execution State. This is implemented as an HCPN by "copying" (obtaining and putting back) the Impression and Status values (i',s') from the Fusion State GF:State_<L>, where L is the current substack level, i.e. the substack level of the current handler, handler_x. The copied value (i',s') represents the PAM Stack Execution State as it was at the time when the current substack level L was first entered (see below for elaboration). The copied value (i',s') is set as the "new" current PAM Stack Execution State, effectively "overwriting" the current PAM Stack Execution State (i,s).

Specifically, recall that given some depth y, if a new substack level is entered by a handler at this depth y, then Substack Execution State at this depth y, denoted (i_y, s_y), is saved in in a place belonging to the fusion set GF:State_<L_y>, where L_y denotes the substack level at the handler depth y. In effect, entering a new substack level at depth y, saves the current PAM Stack Execution state in GF:State_<L_y>. Then, supposing that after entering the substack level L_y at depth y, we execute some handler sequence arriving at the current handler, handler_x, without "leaving" the substack level L_y (i.e.

the sequence of handlers executed after handler_y, but before handler_x have all had their substack levels equal to or higher than L_y), and the Action being executed is "reset", then this Action "resets" the current PAM Stack Execution State back to the state (i_y, s_y).

The next handler to be executed is the next handler in the handler sequence, i.e. handler_ x+1, where x is the current handler depth.

HCPN Templates for Action 'jump'

There are three sub-cases for the pamtester-fm HCPN module templates for Action 'jump'. These are:

- ACTION_JUMP_NEGATIVE,
- ACTION_JUMP_TOO_LONG, and
- ACTION_JUMP.

HCPN module JUMP_NEGATIVE

A jump that is "negative" occurs when the jump value found in the Linux-PAM Configuration is negative and is outside the range [-1,-5] (see below for example). The HCPN Template for ACTION_JUMP_NEGATIVE (implements Action 'jump' for a "negative jump"), is shown in Figure 54.

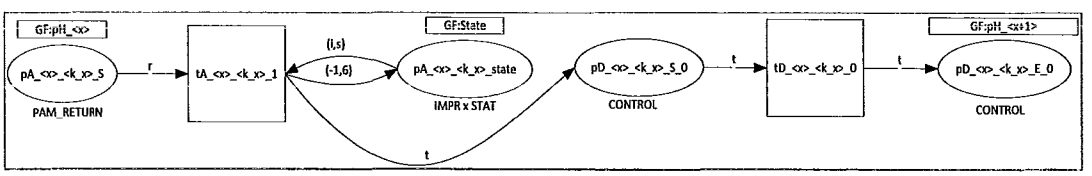

Figure 54: Pamtester-fm HCPN module ACTION_JUMP_NEGATIVE template

The PAM Stack Execution State is changed as follows: Impression is set to _PAM_NEGATIVE = -1, Status is set to _PAM_MUST_FAIL_CODE = _PAM_PERM_DENIED = 6.

The next handler to be executed is the next handler in the sequence, i.e. x+1, where x is the current handler depth. For example, consider the first Configuration Line

```
auth requisite pam_securetty.so
```

contained in the Linux-PAM Configuration in Figure 92 on Page 183 . Suppose that the Linux-PAM Administrator specifies this (erroneous) Configuration Line instead:

```
auth [success=ok default=-7] pam_securetty.so
```

In this case, the $C_0(x) = -7, \forall x \in \{1, 2, .., 30\}$, i.e. if pam_securetty.so PAM returns any error other than 31, then the Action carried out is -7, which is interpreted by Linux-PAM as a "bad jump" of type "negative jump". In this case, the corresponding HCPN Instance is shown in Figure 55.

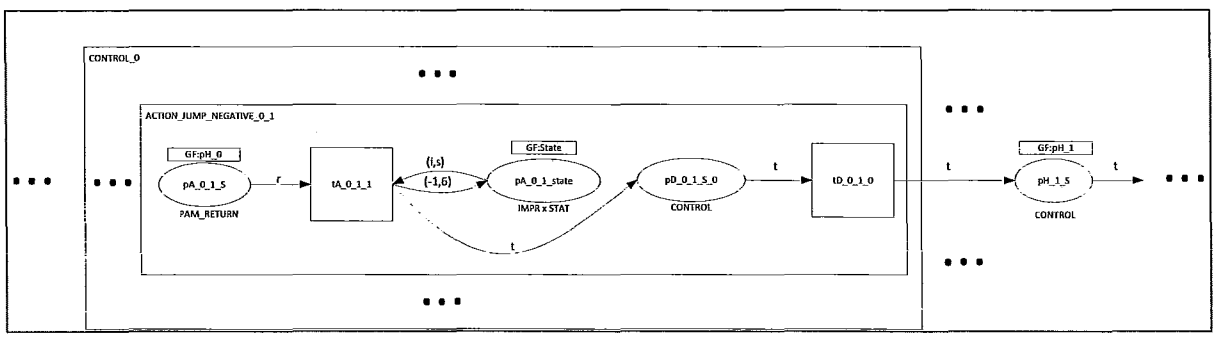

Figure 55: Example of a "bad jump" of type "negative jump"

Note that in this case, the next handler to be executed is the second handler, Handler_1.

HCPN module JUMP_TOO_LONG

A jump that is "too long" occurs as follows. Suppose the PAM Stack Instance contains n handlers (handler_0, handler_1, …, handler_n-1). Also, suppose that the current handler is at depth x, i.e. handler_x, $0 \leq x \leq n-1$. Also, suppose that the substack level of the current handler is L. Further, suppose that the jump specification is the integer $J > 0$. Then, Linux-PAM determines if this Action "jump" is "too long" as follows.

First, obtain the the longest contiguous sequence of handlers handler_x+1, handler_x+2, … , handler_x+y, such that for each handler_z, $z \in x+1 \ldots x+y$, the substack level of handler_z is greater or equal to L, where $x+y < n$ (i.e. we do not consider the termination handler, handler_n).

Second, let K be the number of handlers within this sequence handler_x+1, …, handler_x+y, whose Substack Level equals L.

Third, if $J > K$, then the Action "jump" is "too long". Otherwise, the Action "jump" is a "good jump".

If Linux-PAM determines that the Action "jump" is "too long", then, similarly to the "jump negative", the PAM Stack Execution State is set to (-1,6), i.e. Impression is set to _PAM_NEGATIVE = -1, and Status is set to PAM_MUST_FAIL_CODE = PAM_PERM_DENIED = 6.

In contrast to "jump negative", the next handler to be chosen for execution is the first handler whose depth is greater than x+y, and whose substack level is less than or equal to L (we still skip a contiguous sequence of non-terminal handlers (handlers which are not the terminate handler) whose substack levels are higher than the current substack before picking our next handler to execute). If no such non-terminal handler exists, then the next handler to be executed is the Terminate handler, handler_n.

PAM Stack Instances with a single substack level result in the next handler to be executed being the Terminate handler, handler_n.

The HCPN Template for Action 'jump too long' is shown in Figure 56. The pamtester-fm tool uses this HCPN module ACTION_JUMP_TOO_LONG template, to generate HCPN module ACTION_JUMP_TOO_LONG instances.

The calculation for whether or not an Action "jump" is "too long" is done by the pamtester-fm tool during the parsing of the PAM Stack Instance. We denote this calculation by the function jump_too_long(x). Once an Action is determined to be "too long", the HCPN module ACTION_JUMP_LONG instance is generated, and the next handler to be executed is defined in this instance using the Fusion Set tag template "GF:pH_<jump_too_long(x)>"="GF:pH_<x+y+1>" (as discussed above).

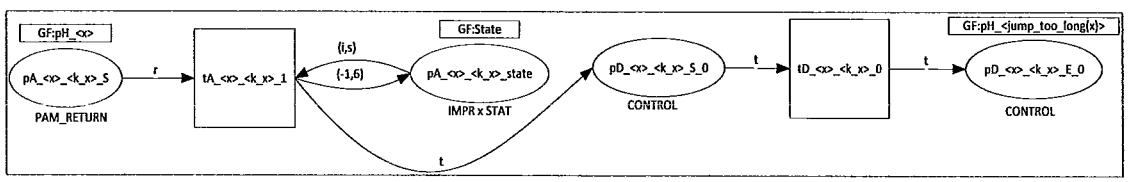

Figure 56: Pamtester-fm HCPN module ACTION_JUMP_TOO_LONG template

For example, consider the first Configuration Line:

```
auth requisite pam_securetty.so
```

contained in the Linux-PAM Configuration in Figure 92 on Page 183 . Suppose that, instead, the Linux-PAM Administrator makes a mistake and specifies this Configuration Line as:

```
auth [success=ok default=13] pam_securetty.so
```

In this case, the $C_0(x) = 13, \forall x \in \{1, 2, .., 30\}$, i.e. if pam_securetty.so PAM returns any error other than 31, then the Action carried out is 13, which is interpreted by Linux-PAM as a "bad jump" of type "jump too long" (since there is less than 13 handlers on the current substack level). In this case, the relevant portion of the corresponding "flattened" HCPN Instance is shown in Figure 57.

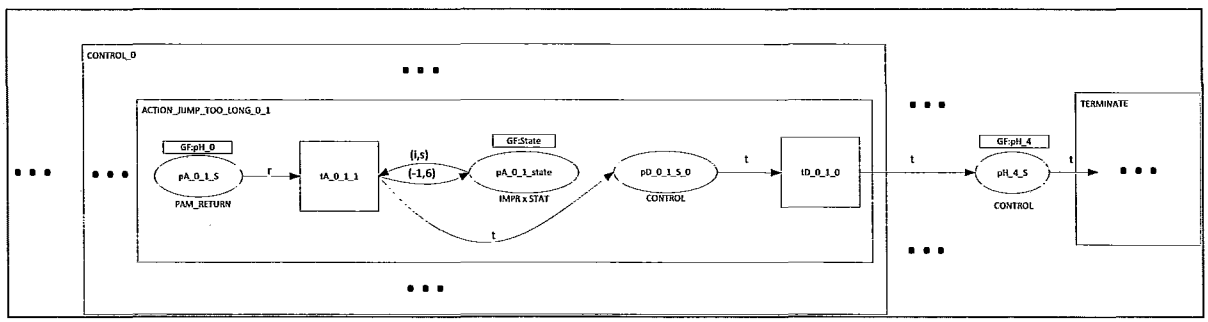

Figure 57: Example of HCPN module ACTION_JUMP_TOO_LONG instance

Note that in this example, the "jump too long" error causes the control of PAM Stack execution to execute the Termination handler as the next handler to be executed. In other words, the PAM Stack Execution is being terminated.

HCPN module ACTION_JUMP

An Action 'jump' that is neither "negative" nor "too long" is considered a "good" jump. Such jumps are represented by the HCPN module ACTION_JUMP template, as shown in Figure 58.

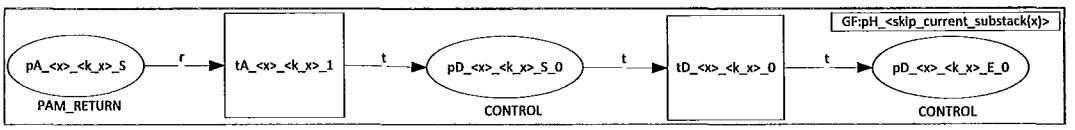

Figure 58: Pamtester-fm HCPN module ACTION_JUMP template

Action 'jump' does not update PAM Stack Execution State.

The handler to be executed next is calculated as follows. Suppose the PAM Stack Instance contains n handlers (handler_0, handler_1, ..., handler_n-1). Also, suppose that the current handler is at depth x, i.e. handler_x, 0<=x<=n-1. Also, suppose that the substack level of the current handler is L. Further, suppose that the jump specification is the integer J > 0.

First, obtain the the longest contiguous sequence of handlers handler_x+1, handler_x+2, ... , handler_x+y, such that for each handler_z, z member of x+1...x+y, the substack level of handler_z is greater or equal to L, and x+y < n (i.e. we do not consider the termination handler, handler_n).

Second, let K be the number of handlers within this sequence handler_x+1, ..., handler_x+y, whose Substack Level equals L.

Third, we assume that J <= K (since otherwise, this Action 'jump' would be "too long").

Fourth, obtain the depth of the K^{th} handler, which we denote by depth(K). Now, depth(K) <= x + y. In other words, going to the handler at depth(K) "jumps" the first K-1 handlers whose substack levels are equal to the current substack level L (and all of the substack handlers in-between whose substack levels are higher than the current substack level L).

Fifth, consider the sequence of handlers, handler_depth(K)+1, …, handler_x+y-1, handler_x+y+1 (here we consider the termination handler, if x+y=n). Choose the first handler, denoted handler_z, depth(K)+1 <= z <= x+y, such that substack level of handler_z <= L. In other words, we still skip all handlers that have higher substack levels, before we finally choose our next handler to be executed.

We denote the above algorithm by the function skip_current_substack(). Skip_current_substack() obtains the current depth x as input, and outputs the depth of the next handler to be executed, as explained in the above algorithm.

Example of "Partial" Unfolding of HCPN Model: HCPN model for ACME Corp

In this section, we illustrate how we use the above-presented HCPN models to create a "partial" unfolding. Our example shows the HCPN model generated by pamtester-fm for the PAM Stack Instance of ACME Corp corresponding to the pam_authenticate()

Management Function for the Service "login". Hence, this Instance is used for the authentication-related functionality of authentication, while using the login program, for example. This PAM Stack instance is shown in Figure 59. It contains four "stacked" PAMs: pam_secure_tty.so, pam_env.so, pam_unix.so and pam_deny.so. Since this PAM Stack Instance corresponds to the pam_authenticate() Management Function, the handler corresponding to each PAM "stacking" contains a pointer to the implementation of the pam_sm_authenticate() Module API function (column entitled int (*func) in Figure 59, i.e. I(pam_sm_authenticate(), pam_securetty), I(pam_sm_authenticate(), pam_env.so), I(pam_sm_authenticate(), pam_unix.so), I(pam_sm_authenticate(), pam_deny.so).

$depth;$ i		$level;$ L_i	$SERVICE$ S_i	$SERVICE$ $GROUP;$ G_i	$CONTROL;$ C_i			$PATH;$ P_i	$OPTIONS;$ O_i
					$X \subseteq R(C_i)$	$d \in D(C_i)$: $\forall x \in X$: $C_i(x) = d$			
	int handler _type	int stack_ level			int actions[32]		int (*func)	char *mod_ name	char **argv
0	0	0	login	auth	0,12 25 "ow"	-1 0 -4	I(pam_sm_authenticate(), pam_securetty.so)	pam_ securetty.so	
1	0	0	login	auth	0,12 "ow"	-1 0	I(pam_sm_authenticate(), pam_env.so)	pam_ env.so	
2	0	0	login	auth	0,12 "ow"	-2 0	I(pam_sm_authenticate(), pam_unix.so)	pam_ unix.so	
3	0	0	login	auth	0,12 25 "ow"	-1 0 -3	I(pam_sm_authenticate(), pam_deny.so)	pam_ deny.so	

Figure 59: Generation of a PAM Stack instance for the pam_sm_authenticate() Module API function for theService "login"

Recall that the execution of a PAM Stack instance, as per algorithm shown in Table 13 on page 29, executes some handler subsequence. The executed handler subsequence generates an authentication-related functionality instance. Given a handler execution subsequence, each handler execution causes the execution of its corresponding Management Function implementation. Each Management Function implementation execution contributes to the provision of the authentication-related functionality of the PAM Stack Instance as a whole. Depending on the PAM_RETURN value obtained from

each handler execution, the appropriate Action is executed. The execution of each Action may update the PAM Stack Execution State, as well as chooses the next handler to be executed.

Recall that pamtester-fm generates an HCPN model that describes the above process. Specifically, the generated HCPN model "captures" all of the possible handler execution sub-sequences. In other words, the HCPN model captures all of the possible authentication-related functionalities that can be provided by the corresponding PAM Stack Instance.

Returning to our example, in this particular "partial" unfolding, for each PAM, we define the set of PAM_RETURN values that the PAM is capable of returning in Table 14. This list of return values is not complete. This table is created for illustration purposes only. In fact, some of these PAMs are capable of returning additional PAM_RETURN values.

PAM	Possible PAM_RETURN values of the PAM's implementation of pam_sm_authenticate()
pam_securetty.so	PAM_SUCCESS = 0 PAM_SERVICE_ERR = 3 PAM_AUTH_ERR = 7 PAM_IGNORE = 25 PAM_INCOMPLETE = 31
pam_env.so	PAM_SUCCESS = 0 PAM_BUFF_ERR = 7 PAM_IGNORE = 25 PAM_ABORT = 26
pam_unix.so	PAM_SUCCESS = 0 PAM_IGNORE = 25 PAM_INCOMPLETE = 31
pam_deny.so	PAM_AUTH_ERR = 7

Table 14: Possible PAM_RETURN values of PAMs "stacked" on ACME Corp's pam_authenticate() PAM Stack Instance

Finally, in Figure 60 we show the HCPN model of all of the possible executions of the PAM Stack Instance of ACME Corp. This HCPN model is a "partial" unfolding comprised of an HCPN module INITALIZE instance, four HCPN module HANDLER instances, and a TERMINATE instance. We also add labelled "containers" for the purpose of visual presentation. These containers denote portions of the HCPN model that correspond to HCPN modules HANDLER_<x>, MODULE_<x>, CONTROL_<x>, ACTION_<x>_<k_x>.

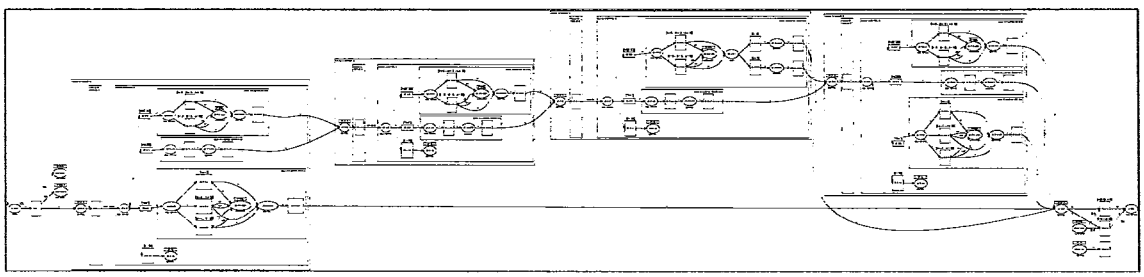

Figure 60: HCPN model - ACME Corp

The HCPN modules INITIALIZE and HANDLER_0 are shown in Figure 61.

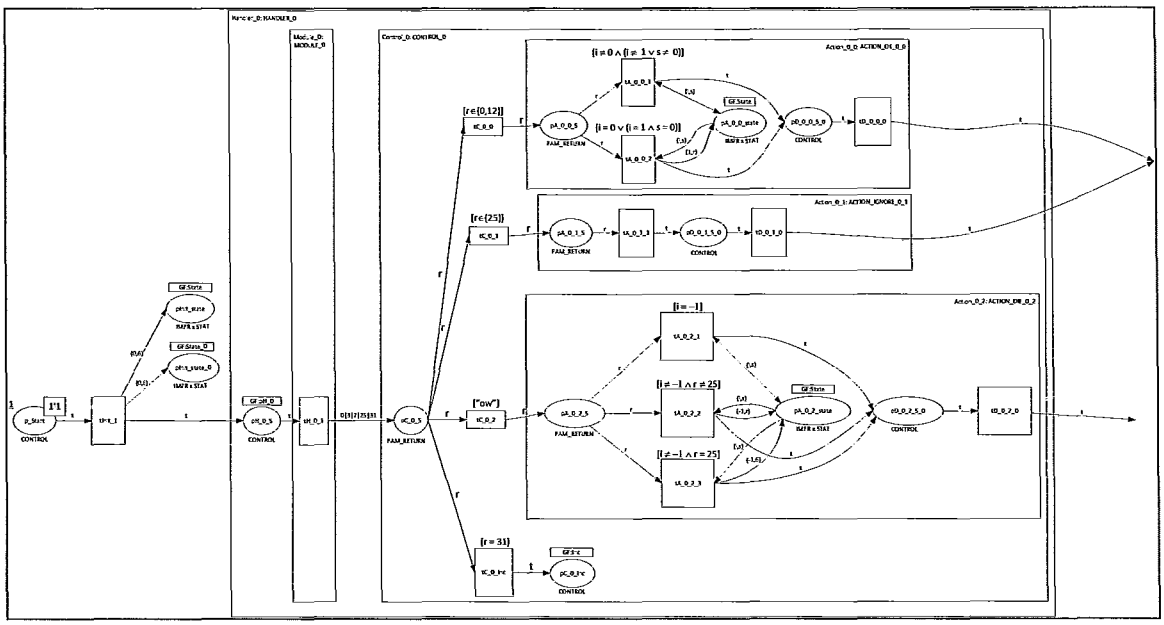

Figure 61: Close-up of HCPN modules INITIALIZATION and HANDLER_0

The HCPN module HANDLER_1 is shown in Figure 62.

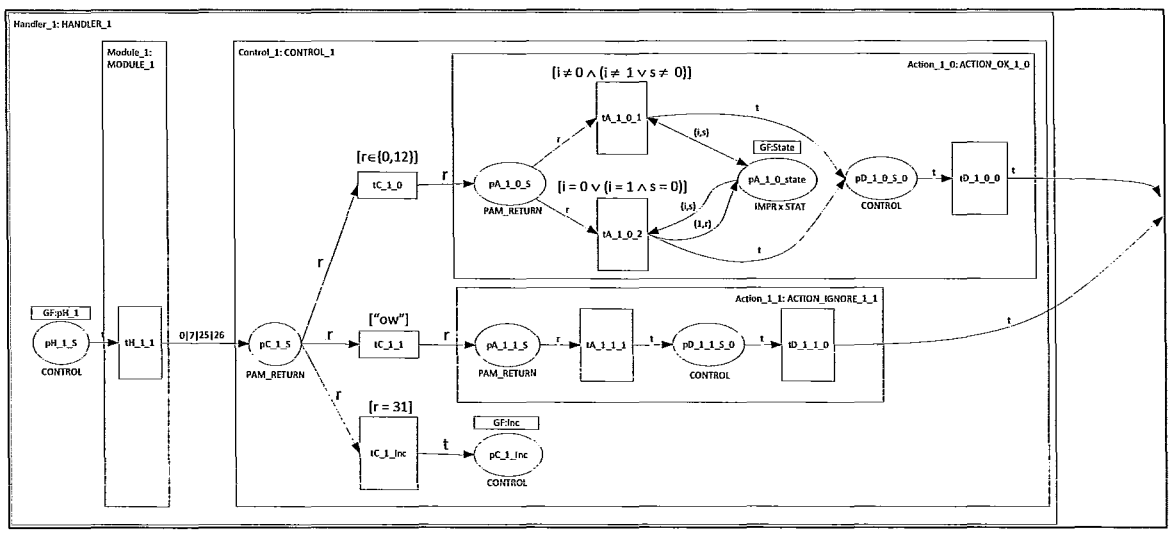

Figure 62: Close-up of HCPN module HANDLER_1

The HCPN module HANDLER_2 is shown in Figure 63.

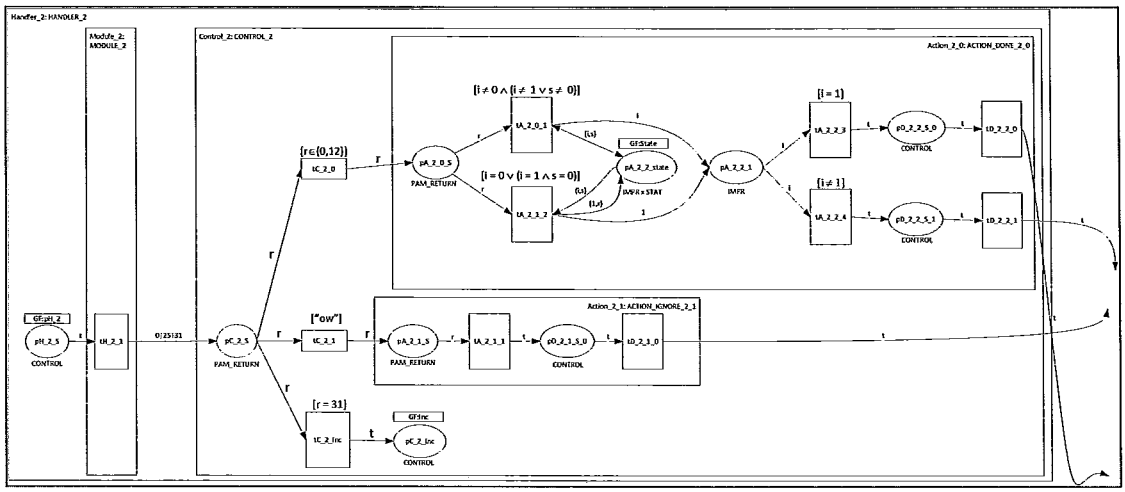

Figure 63: Close-up of HCPN module HANDLER_2

The HCPN modules HANDLER_3 and TERMINATE are shown in Figure 64.

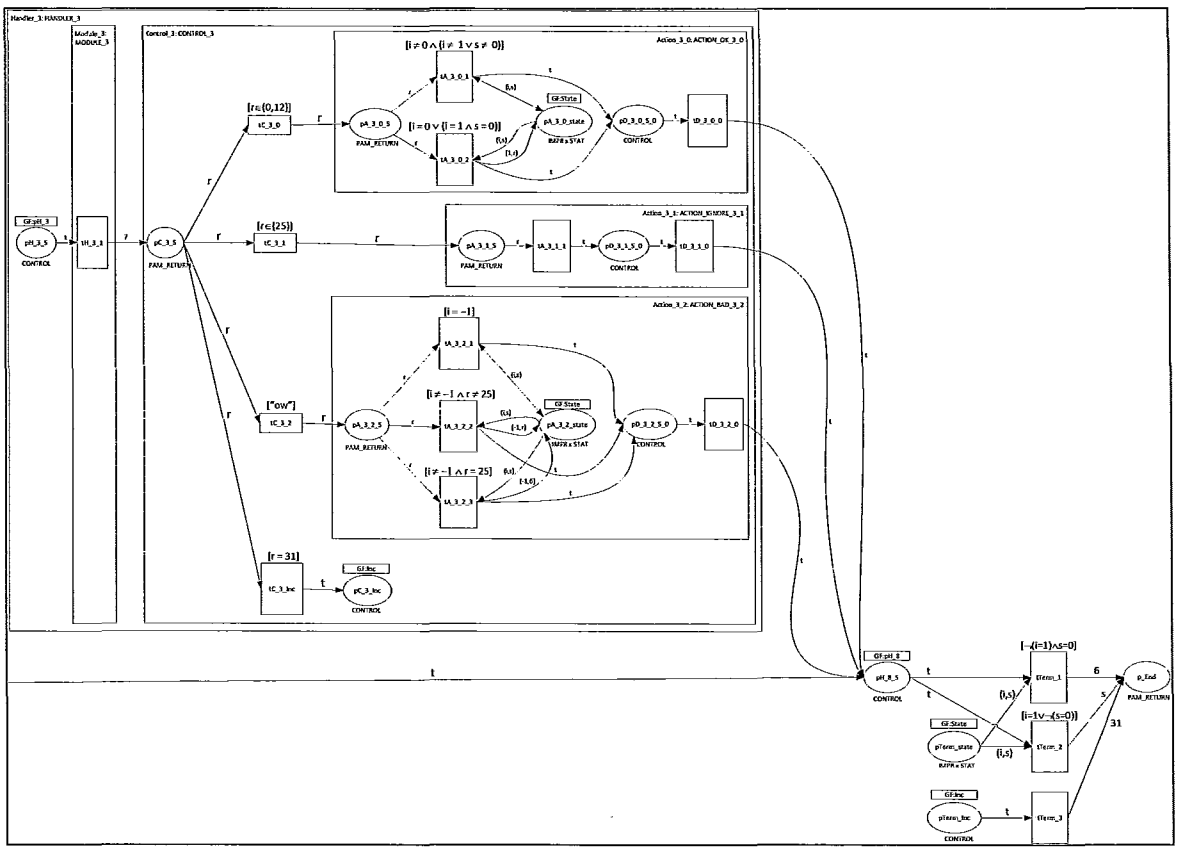

Figure 64: Close-up of HCPN module HANDLER_3

The Instance Hierarchy describing this HCPN model is shown in Figure 65.

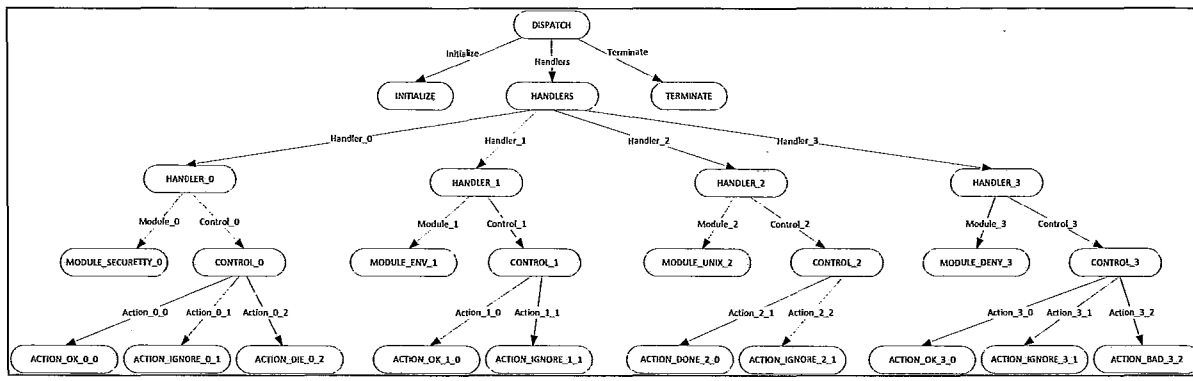

Figure 65: Instance Hierarchy of HCPN model

METHODOLOGY – PART II: Transition System Modeling

Introduction to Transition System Modeling

A Transition System is a mathematical formalism that specifies how a system behaves. We use Transition Systems to specify how our HCPN models behave. Pamtester-fm generates Transition System specifications based on the corresponding HCPN specifications. Pamtester-fm uses NuSMV syntax to specify these Transition Systems.

Once the Transition System specification is generated, Pamtester-fm executes NuSMV. During this execution, pamtester-fm supplies NuSMV with: Transition System syntax, and "security properties" to check on this Transition System. Then, Pamtester-fm directs NuSMV to build the Transition System models based on this syntax, and instructs NuSMV to model check the resulting Transition System models.

The results of this model checking are then obtained and interpreted by pamtester-fm. These results describe the possible behaviours of the corresponding HCPN models, which in turn, describe the possible authentication-related functionalities of the PAM Stack Instances. In other words, these results inform us about the authentication-related functionalities that can be produced by the PAM Stack Executions of the corresponding PAM Stack Instances.

We used NuSMV for two reasons. First, NuSMV can be programmatically supplied with arbitrary Transition System model specifications, as well as be programmatically controlled to then build and model check these models. Second, one goal of this thesis was to provide groundwork for an open source tool which can be used by Linux-PAM Administrators. Such a tool must be easy to use. Ideally, such a tool should operate with

minimal interaction from its operator. Additionally, such a tool must not require the operator to have knowledge of formal specification or verification, including HCPNs, Transition Systems and the method of model checking. Pamtester-fm achieves both of these goals by being automated, while hiding the details of formal model specification and verification.

Transition Systems

Transition Systems allow us to describe a system in terms of its states, and how the system changes between states. For example, the system in Figure 66 has three states s_1, s_2 and s_3. Furthermore, this system can change states as follows: from s_1, the system can change to state s_2 or state s_3. From state s_3,the system can change to state s_1. From state s_2, the system can change to state s_2.

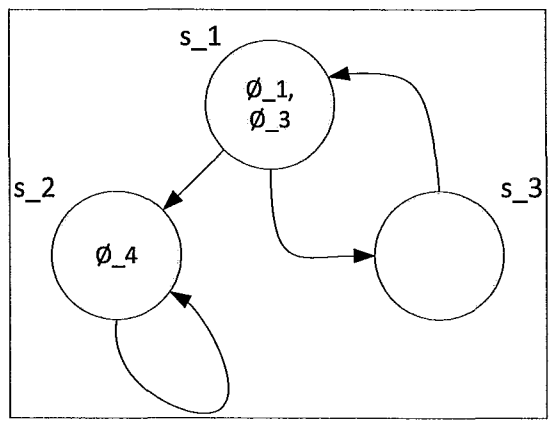

Figure 66: Example of Transition System

A Transition System also has a number of properties. Depending on the state that the system is in, a system's property can be either true or false in this state. For example,

suppose that the Transition System in Figure 66 has the following properties: ϕ_1, ϕ_2, ϕ_3, ϕ_4, ϕ_5. Then, a property is true in a given state, if, and only if this property is shown inside the circle representing the state. For example, in state s_1, ϕ_1 and ϕ_3 are true, and ϕ_2, ϕ_4, ϕ_5 are false.

Formal Definition of a Transition System

Formally, as defined in (6), a Transition System is a three-tuple (S, -> L) along with a set of atomic expressions called Atoms. Atomic expressions evaluate to either true or false, depending on the state that the system is currently in. For example, in Figure 66, Atoms = {ϕ_1, ϕ_2, ϕ_3, ϕ_4, ϕ_5}. S is a set of elements called states. For example, in Figure 66, S = {s_1, s_2, s_3}. The symbol -> denotes a total binary relation on S, i.e. -> $\subseteq S \times S$ such that $\forall s \in S \exists t \in S$: s -> t. Essentially, -> specifies how the system can change states. Given two states x, y in S, x->y iff S can change from state x to the state y. For example, in Figure 66, s_1->s_2, s_1->s_3, s_3->s_1, and s_2->s_2. L is a labelling function L:S->P(Atoms) where P denotes a power set. Given a state s, and an atomic expression e belonging to Atoms, $e \in L(s)$ iff e is true in the state s. For example, in Figure 66, L(s_1)={ϕ_1, ϕ_3}, L(s_2)={ϕ_4}, L(s_3)={ }.

The State Space Explosion Problem

In general, the size of the set of states S of a Transition System (S,->,L) depends on the number of variables of the system and the domains of each of these variables. For example, consider a system whose variables can be expressed by two variables, x and y,

where each variable can have a value of 0 or 1. Further, suppose that we decide to represent this system in terms of the possible values of these variables as two tuples (value(x), value(y)), where value(x), value(y) is the value of variable x, y, respectively. Then, all possible states of this system can be described as S = { (0,0), (0,1), (1,0), (1,1) }.

Now, if a third variable z is added to create a new system, and supposing that the domain of variable z is {0,1,3}, then, following the same approach, the set of states of this system S = { (0,0,0), (0,1, 0), (1,0,0), (1,1,0), (0,0,1), (0,1,1), (1,0,1), (1,1,1), (0,0,3), (0,1,3), (1,0,3), (1,1,3) }. Thus, the state space of the system was multiplied by three, with the addition of the new variable z.

In general, a variable, when added to a system description, multiplies the number of states of the system. Thus, the number of states of the system grows exponentially in the number of variables of the system. This is called the *state space explosion* problem. Essentially, the number of variables and the domains of these variables cause the number of states of the system to become too large, even for automated, computer-based tools to handle. Systems suffering from the state space explosion problem require too much CPU time or too much RAM for analysis purposes.

When designing transition systems, one must take care to ensure that the state space explosion problem does not occur. In real-world applications, the state space explosion problem is common.

Connection between HCPNs and Transitions Systems

Given an HCPN model generated by pamtester-fm, we encode the behaviour of this HCPN as a Transition System. Specifically, we define S to be the set of markings of the HCPN that are reachable from the initial marking. We define the Transition Relation -> to be the Firing Rule, where given two markings s_i, s_j, s_i -> s_j iff there exists a firing of a single HCPN transition that causes the marking s_i to become the marking s_j. We define the set of Atoms as a set of NuSMV expressions. The terms of these expressions are dependent on the markings of the HCPN. For example, we define the expression "p_End = 0", where p_End is an HCPN place and 0 is a member of the colour set associated with p_End. Then, we define "p_End = 0" to evaluate to TRUE iff the token of value 0 is in the place p_End.

For example, consider the HCPN shown in Figure 67.

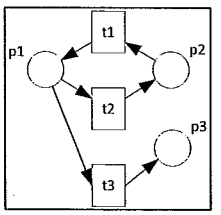

Figure 67: HCPNs as Transition Systems – Example

Now, consider the following markings, say, marking M_1, marking M_2, and marking M_3, shown in Figure 68, Figure 69, Figure 70, respectively.

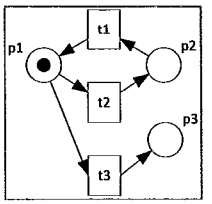

Figure 68: HCPNs as Transition Systems – Example – Marking M_1

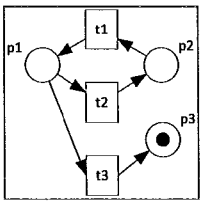

Figure 69: HCPNs as Transition Systems – Example – Marking M_2

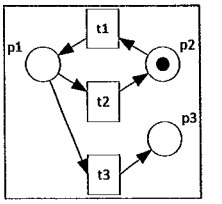

Figure 70: HCPNs as Transition Systems – Example – Marking M_3

Now, given the marking M_1, according to the Firing Rule, both transitions t2 and t3 are enabled. Thus, either one of the transitions can fire (but not both, since there is only a single token available for consumption in place p1). If transition t3 fires, then the HCPN changes its marking from M_1, to the marking M_2. In a sense, we can interpret this as the HCPN changing its state from M_1 to M_2.

If the HCPN is in the marking M_2, then the HCPN is in a "deadlock" state. This means that the HCPN cannot change its state anymore, since no transition can be enabled by the Firing Rule. Thus, no transition can fire. Thus, no new marking can be obtained. A deadlock state can be interpreted as the HCPN staying in the same marking, in this case, marking M_2, indefinitely. Another way to look at this is that the HCPN changes its marking from marking M_2 to marking M_2 indefinitely.

On the other hand, if the HCPN is in state M_1, and transition t2 is fired, then the HCPN changes from M_1 to M_3. Once the HCPN is in M_3, the only state change that can occur is to go from M_3 back to M_1.

We can illustrate the above "behaviour" of the HCPN as shown in Figure 71, via a graph-like structure.

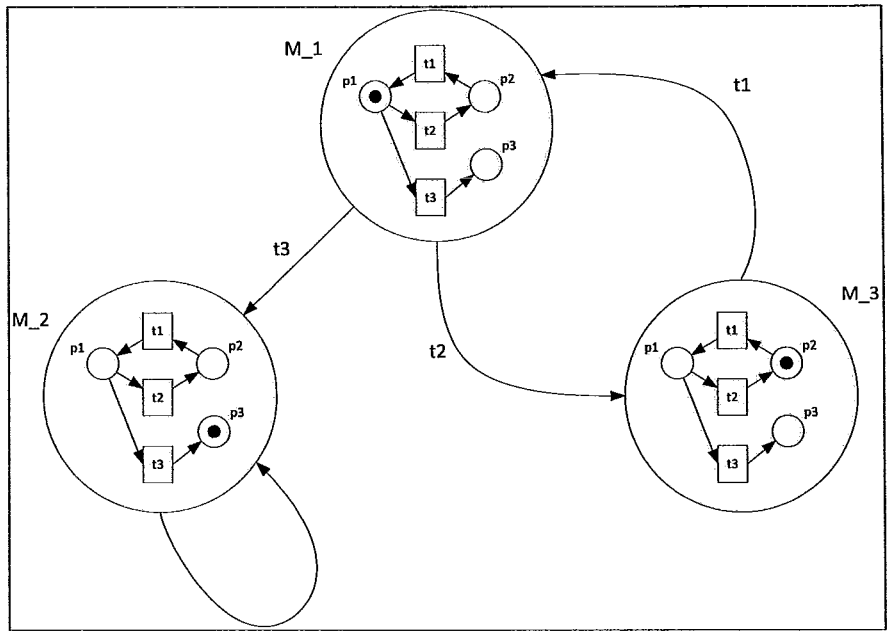

Figure 71: HCPN Behaviour as a Transition System

Each node of this graph (a large circle containing a Petri Net) represents a state of the Transition System, and corresponds to a single HCPN marking. This is why we define the states of the Transition Systems as HCPN markings. For our HCPN models, the changing from a marking to another marking corresponds to a unique transition being fired. Hence, we associate such edges with the name of the fired transitions. For example, in Figure 71, the edge outgoing from the node associated with the marking M_1, and the node associated with the marking M_2 is labelled t3. This is because it was the firing of t3 that caused the HCPN to change its state from marking M_1 to marking M_2.

Precisely, a node M_i is connected with an outgoing arrow from M_i to a node M_j, $i \neq j$, where the arrow is incoming to node M_j iff the marking M_i can change to the marking M_j via a firing of a single transition belonging to the HCPN.

One exception is the state M_2. M_2 has a "loop" edge from M_2 to M_2. In this case, the "loop" edge does not correspond to any transition firing. In this case, the underlying HCPN is in a deadlock state (there are no enabled transitions, hence no transition can fire, hence the HCPN cannot change its marking anymore, i.e. is in a deadlock). In fact, due to the structure of the HCPNs generated by pamtester-fm, the only deadlock state that can be achieved by an HCPN of an arbitrary PAM Stack Instance is when there is a token in the "termination" place of the HCPN, place p_End. In this case, we ensure, via our definition of the transition relation ->, that a "deadlock" transition, represented in the graph as a "loop" edge, is defined.

Note that the "Transition System" presented in Figure 71 is not exactly (formally) correct. It was presented here in this way to not complicate the issue, and to show the link between HCPN markings, and how the changes in the HCPN markings define a "Transition System-like" structure. There exists another Transition System state to which the state M_2 changes to. Furthermore, in the (formally correct) Transition System, the "loop" edge of M_2 does not exist. This is elaborated on when we discuss the definition of a Transition Relation (page 126).

In general, given an HCPN model generated by pamtester-fm, for an arbitrary PAM Stack Instance, when we generate the corresponding Transition System, we restrict the Firing Rule to only be able to fire a single transition at a time. This is done by the particular way in which we define the Transition Relation in NuSMV syntax. This is also due to the way that NuSMV transitions between states.

This does not limit the "expressive" power of our model. This is because an arbitrary firing instance, consisting of an arbitrary multiset of fired transitions can be equivalently represented by a sequence of single transition firings (5).

Approach for NuSMV Encoding of HCPN Behaviour

Introduction

Similar to HCPN generation, pamtester-fm does automatic generation of Transition Systems. Just like HCPN generation, Transition System generation is based on templates. The generated Transition System is represented by a NuSMV encoding. This NuSMV encoding is a text file containing NuSMV syntax. This NuSMV syntax encoding describes a Transition System: a set of states S, a transition relation ->, and a labelling function L.

HCPN places are encoded as NuSMV variables, and hence, contribute to the State Space size. In contrast, HCPN transitions are not encoded as NuSMV variables. Thus, HCPN transitions have negligible State Space size contribution. Some additional NuSMV variables are created, where these variables do not correspond to any HCPN places, i.e. NuSMV variable 'gf_depth', i.e. NuSMV variable 'firedTransition'.

Considerable effort was made to minimize the state space of the generated Transition System, as well as to maximize its "human readability" and "similarity" with its corresponding HCPN. Structural similarity between an HCPN and its corresponding Transition System was kept as close as possible, while still allowing for significant State Space minimization.

State Space Minimization

In terms of minimizing the size of the state space, the following was done:

- re-use of commonly used sets of related variables, and
- minimization of domains of variables corresponding to HCPN places.

Specifically, a critical state space size minimization was achieved by re-using commonly used sets of related variables. For instance, the variables used to represent all HCPN module Action instances, based on ACTION_<name(x)>_<x>_<k_x> templates, were re-used. For example, in the HCPN model for ACME Corp (Figure 65, 99), there are three Action 'ok' HCPN module instances used: ACTION_OK_0_0, ACTION_OK_1_0, and ACTION_OK_3_0. In this case, only one set of NuSMV variables is used to represent all three of these HCPN module instances.

Also, since we generate the transition system based on an (already existing) HCPN model instance, we know all possible tokens that can be placed in each HCPN place. Due to this, for some HCPN places, the corresponding NuSMV variables are generated based on a template, where this template includes procedures for minimizing NuSMV variable domains. Specifically, these templates define the variable domain as containing only the values that correspond to the possible HCPN token values. This is done for all HCPN pH_<x> places, since we know ahead of time what possible PAM_RETURN values a particular PAM can generate.

Furthermore, we made use of HCPN fusion sets were it made sense. This way, all HCPN places belonging to the same Fusion Set were defined as the same NuSMV variable.

Combining the above approaches resulted in considerable state space minimization. Table 15 - Table 20 summarize the results of these optimizations on the individual HCPN modules.

HCPN Module	Place	Corresponding NuSMV variable domain size	State Space Contribution (in bits)	
			without optimizations	with optimization
Initialization	p_Start	1	1	1
	pInit_state	4×33	$2 + 6$	0
	pInit_state_0	4×33	$2 + 6$	0
	TOTAL		17	1
Termination	pH_8_S	1	1	1
	pTerm_state	4×33	$2 + 6$	0
	pTerm_Inc	1	1	0
	p_End	32	5	5
	TOTAL		15	6

Table 15: State Space Minimization - Initialization and Termination

These results provide contributions to the state space of the corresponding Transition System by each HCPN module. For each HCPN module type, the places of the module are listed, since it is the places that are represented as NuSMV variables (and hence contribute to the state space size of the Transition System). Also, for each place, the size of the variable domain is shown. For example, given the HCPN module Initialization and the place pInit_state, its variable domain size is 4x33. This denotes the fact that pInit_state has a cross-product set as its domain (its colour set is IMPR x STAT = {EMPTY,-1,0,1} x {EMPTY, 0, 1, ..., 31}), hence, cardinality of domain is 4x33. We do not multiply 4x33 to obtain 132 because we compute the state space contribution on a per set basis, if the set is a cross product. This is because we encoded the colour set IMPR x

STAT by two different NuSMV variables, where one variable encodes IMPR and one variable encodes STAT. Hence, the combined two-NuSMV-variable contribution of the HCPN place pInit_state is: 2 bits for IMPR (base 2 logarithm of size of variable domain, rounded up), and 6 bits for STAT. Then, total contribution is 2+6 bits, or 8 bits for pInit_state. Hence, total contribution of a single instance of the Initialization HCPN module is 17 bits of state space (without any optimizations).

With all of our optimizations, the state space contribution for the HCPN module Initialization is just a single bit (1 bit) – a difference of 16 bits! In terms of the number of states, before optimization, the Initialization HCPN module would require 2^17 = 131072 states. After the optimization, this module only requires 2 states – a difference of 131070 states!

Note that Initialization and Termination HCPN modules are constant in the number of handlers of the PAM Stack Instance. For example, it does not matter how many handlers comprise the PAM Stack instance, i.e. 2, 10, 100, 1000, etc., since the HCPN model will use the Initialization and Termination modules exactly once. Hence, the state space contribution of the Initialization and Termination HCPN module instances is constant in the number of handlers: 17 and 15 bits without optimization, and 1 and 6 bits (with optimization), respectively.

The state space contribution of our other HCPN modules may not be constant in the number of handlers, and in the number of HCPN module instantiations (i.e. the number of distinct times that an HCPN module is used in the HCPN model) . In other words, if the number of handlers increases, and the number of times an HCPN module is used

increases, then so may the state space contribution. We denote the number of handlers comprising the PAM Stack instance by $n, n > 0$. We denote the number of times a particular HCPN module has been instantiated by the HCPN model by k, $k \geq 0$. We can see the use of the n and k parameters in the following tables.

HCPN Module	Place	Corresponding NuSMV variable domain size	State Space Contribution (in bits), n := number of handlers of substack k := number of times the corresponding HCPN module instance is used in the HCPN model	
			without optimizations	with optimization
HANDLER_<x>	pH_<x>_S	2	$1 \times n$	$1 \times n$
MODULE_<x>	N/A	N/A	0	0
MODULE_SUBSTACK	pH_<x>_S	2	$1 \times k$	$1 \times \left\lceil \frac{k}{k+1} \right\rceil$
	pH_<x>_0_state	4×33	$(2+6) \times k$	0
	pH_<x>_0_state_<L+1>	4×33	$(2+6) \times k$	0
	pH_<x>_<x+1>	2	$1 \times k$	0
	TOTAL		$18 \times k$	$1 \times \left\lceil \frac{k}{k+1} \right\rceil$

Table 16: State Space Minimization - HANDLER_<x>, MODULE_<x>, MODULE_SUBSTACK, CONTROL_<x>

For example, a HANDLER_<x> HCPN module instance, before any optimizations, has a state space contribution that is linear in the number of handlers that comprise the stack (although there is only one HANDLER_<x>_HCPN module instance per HCPN model). In this case, the HCPN place pH_<x>_S has a domain CONTROL = {1}. In NuSMV the domain of the corresponding NuSMV variable is encoded as {EMPTY, 1}. Hence, the size of the domain of the NuSMV variable is 2. Now, since, as is shown in the template for the HCPN module HANDLER_<x>, for each handler comprising the PAM Stack Instance, the template generates a place pH_<x>_S. Thus, given n handlers, the

template generates pH_0_S, pH_1_S, …, pH_(n-1)_S – a total of n HCPN places. Since each of these places contributes a single bit (1 bit) to the state space size, hence the total number of bits contributed by the HANDLER_<x> HCPN module is $1 \times n = n$ bits (1 bit for each of the n places). In this case, this contribution stays the same, even if optimizations are used.

The MODULE_<x> HCPN module does not contain any HCPN places (hence N/A, for Not Applicable). Thus, MODULE_<x> contributes zero bits (0 bits) to the state space (with and without optimizations).

For the rest of the HCPN modules, the state space is highly sensitive to the parameters n and k. For example, the PAM Stack Instance for ACME Corp (see

Figure 59 on page 93) uses the HCPN module ACTION_OK three times (see the corresponding Instance Hierarchy in Figure 65 on page 99). Then, without optimizations, each of the three ACTION_OK HCPN module instances (ACTION_OK_0_0, ACTION_OK_1_0, ACTION_OK_3_0) would make their own individual contributions to the state space size. In this case, together, as shown in Table 18, all three ACTION_OK instances make a total contribution of $14 \times k$ bits, which in this example evaluates to $14 \times k = 14 \times 3 = 42$ bits of state space contribution. With optimization, these ACTION_OK HCPN module instances, together, contribute $6 \times \left\lceil \frac{k}{k+1} \right\rceil = 6 \times \left\lceil \frac{3}{3+1} \right\rceil = 6 \times 1 = 6$ bits to the state space. Thus, without optimization, *each* ACTION_OK module instance adds 14 bits of state space or 2^14=16384 states, and with optimization, *all* ACTION_OK module instances only contribute 6 bits or 2^6=64 states – a difference of

16320 states or 8 state space contribution doublings! Moreover, without optimization, the number of contribution bits is linear (i.e. $14 \times k$ is a linear function), whereas $\left\lceil \frac{k}{k+1} \right\rceil = 1, \forall k > 0$, and $\left\lceil \frac{k}{k+1} \right\rceil = 0, k = 0$. In other words, $\left\lceil \frac{k}{k+1} \right\rceil$ is just a way for us to say "if the HCPN module is used at least once (i.e. $k > 0$), then add a constant number of bits, i.e. for $6 \times \left\lceil \frac{k}{k+1} \right\rceil$, add 6 bits of state space to the state space (irrespective of the number of times the HCPN module is used), otherwise, this HCPN module does not contribute to the state space (since it is not used, i.e. $k = 0$).

As shown in Table 17, given an HCPN module CONTROL_<x> instance, its state space contribution, before optimization, is linear in the number of instances, i.e. $7 \times k$. With optimization, contribution of each CONTROL_<x> is $\lceil log_2(|R_x|) \rceil \in [1,6]$, where R_x is the set of possible PAM_RETURN values of HCPN module MODULE_<x> (see page 67). This is the optimization where we set the NuSMV variable corresponding to the HCPN place pC_<x>_S to contain, and only contain, the values corresponding to the possible PAM_RETURN values of the preceding MODULE_<x> HCPN module. Specifically, R_x is the set of PAM_RETURN values that can be returned by the execution of the PAM's implementation of the Management Function $I(f^{sm}, P_x)$ associated with the Effective PAM Stack instance Π^f. The number of possible PAM_RETURN values is between 1 and 32 (at least one member of {0,1,…,31} must be returned by a PAM's Management Function implementation $I(f^{sm}, P_x)$). Lastly, we must account for the EMPTY value, representing no tokens in place pC_<x>, thus pC_<x> can contain at most 33 values. For k CONTROL_<x> instances, we obtain $k \times \lceil log_2(|R_x|) \rceil \in [1 \times k, 6 \times k]$.

HCPN Module	Place	Corresponding NuSMV variable domain size	State Space Contribution (in bits), k := number of times the corresponding HCPN module instance is used in the HCPN model	
			without optimizations	with optimization
CONTROL_<x>	pC_<x>_S	33	$6 \times k$	$[1 \times k, 6 \times k]$
	pC_<x>_Inc	2	$1 \times k$	0
	TOTAL		$7 \times k$	$[1 \times k, 6 \times k]$

Table 17: State Space Minimization - CONTROL_<x>

The next table, Table 18 shows the state space contributions of HCPN module ACTION_<name(x)> instances. Again, without any optimizations, contributions of HCPN module ACTION_<name(x)> instances are linear in the number of times they are used in the PAM Stack Instance. In contrast, with optimization, the state space contribution of these instances is constant!

HCPN Module	Place	Corresponding NuSMV variable domain size	State Space Contribution (in bits), n := number of handlers of substack k := number of times the corresponding HCPN module instance is used in the HCPN model	
			without optimizations	with optimization
ACTION_IGNORE	pA_<x>_<k_x>_S	32	$5 \times k$	$5 \times \lceil \frac{k}{k+1} \rceil$
	pD_<x>_<k_x>_S_0	2	$1 \times k$	$1 \times \lceil \frac{k}{k+1} \rceil$
	pD_<x>_<k_x>_E_0	2	$1 \times k$	0
	TOTAL		$7 \times k$	$6 \times \lceil \frac{k}{k+1} \rceil$
ACTION_OK ACTION_BAD ACTION_DIE	pA_<x>_<k_x>_S	32	$5 \times k$	$5 \times \lceil \frac{k}{k+1} \rceil$
	pA_<x>_<k_x>_state	4×33	$(2+6) \times k$	0
	pD_<x>_<k_x>_S_0	2	$1 \times k$	$1 \times \lceil \frac{k}{k+1} \rceil$
	pD_<x>_<k_x>_E_0	2	$1 \times k$	0
	TOTAL		$15 \times k$	$6 \times \lceil \frac{k}{k+1} \rceil$
ACTION_DONE	pA_<x>_<k_x>_S	32	$5 \times k$	$5 \times \lceil \frac{k}{k+1} \rceil$
	pA_<x>_<k_x>_state	4×33	$(2+6) \times k$	0
	pA_<x>_<k_x>_1	3	$2 \times k$	$2 \times \lceil \frac{k}{k+1} \rceil$
	pD_<x>_<k_x>_S_0	2	$1 \times k$	$1 \times \lceil \frac{k}{k+1} \rceil$
	pD_<x>_<k_x>_S_1	2	$1 \times k$	$1 \times \lceil \frac{k}{k+1} \rceil$
	pD_<x>_<k_x>_E_0	2	$1 \times k$	0
	pD_<x>_<k_x>_E_1	2	$1 \times k$	0
	TOTAL		$19 \times k$	$9 \times \lceil \frac{k}{k+1} \rceil$
ACTION_RESET	pA_<x>_<k_x>_S	32	$5 \times k$	$5 \times \lceil \frac{k}{k+1} \rceil$
	pA_<x>_<k_x>_state	4×33	$(2+6) \times k$	0
	pA_<x>_<k_x>_state_s	4×33	$(2+6) \times k$	0
	pD_<x>_<k_x>_S_0	2	$1 \times k$	$1 \times \lceil \frac{k}{k+1} \rceil$
	pD_<x>_<k_x>_E_0	2	$1 \times k$	0
	TOTAL		$23 \times k$	$6 \times \lceil \frac{k}{k+1} \rceil$

ACTION_JUMP	pA_<x>_<k_x>_S	32	$5 \times k$	$5 \times \left\lceil \frac{k}{k+1} \right\rceil$
	pD_<x>_<k_x>_S_0	2	$1 \times k$	$1 \times \left\lceil \frac{k}{k+1} \right\rceil$
	pD_<x>_<k_x>_E_0	2	$1 \times k$	0
	TOTAL		$7 \times k$	$6 \times \left\lceil \frac{k}{k+1} \right\rceil$
ACTION_JUMP _NEGATIVE	pA_<x>_<k_x>_S	32	$5 \times k$	$5 \times \left\lceil \frac{k}{k+1} \right\rceil$
	pA_<x>_<k_x>_state	4×33	$(2+6) \times k$	0
	pD_<x>_<k_x>_S_0	2	$1 \times k$	$1 \times \left\lceil \frac{k}{k+1} \right\rceil$
	pD_<x>_<k_x>_E_0	2	$1 \times k$	0
	TOTAL		$15 \times k$	$6 \times \left\lceil \frac{k}{k+1} \right\rceil$
ACTION_JUMP _TOO_LONG	pA_<x>_<k_x>_S	32	$5 \times k$	$5 \times \left\lceil \frac{k}{k+1} \right\rceil$
	pA_<x>_<k_x>_state	4×33	$(2+6) \times k$	0
	pD_<x>_<k_x>_S_0	2	$1 \times k$	$1 \times \left\lceil \frac{k}{k+1} \right\rceil$
	pD_<x>_<k_x>_E_0	2	$1 \times k$	0
	TOTAL		$15 \times k$	$6 \times \left\lceil \frac{k}{k+1} \right\rceil$

Table 18: State Space Minimization - ACTION_IGNORE, ACTION_OK, ACTION_BAD, ACTION_DIE, ACTION_DONE, ACTION_RESET, ACTION_JUMP, ACTION_JUMP_NEGATIVE, ACTION_JUMP_TOO_LONG

The next table shows the state space contribution made by the Fusion Sets of the HCPN Model.

HCPN Module	HCPN Places	Corresponding NuSMV variable domain size	State Space Contribution (in bits), L := maximum substack level of PAM Stack Instance, $0 \leq L \leq 15$	
			without optimizations	with optimization
DISPATCH	GF:State	4×33	N/A	$2+6$
DISPATCH	GF:Inc	2	N/A	1
DISPATCH	GF:State_0, …, GF:State_<L>	$(4 \times 33) \times (L+1)$	N/A	$(2+6) \times (L+1)$

Table 19: State Space Minimization - Fusion Sets

Here, the column "without optimizations" shows Not Applicable (N/A) because we interpret Fusion Sets as optimizations. The HCPN places GF:State_0, GF:State_1, …, GF:State_<L>, $0 \leq L \leq 15$, denote the PAM Stack Execution Substack Level states. Pamtester-fm, during parsing of the PAM Stack Instance, determines the maximum substack level of the handlers comprising the PAM Stack Instance. Hence, the generated HCPN, and the corresponding Transition System only refer to Substack Levels between 0 and the maximum Substack Level detected. This maximum stubstack level is denoted by L.

Lastly, Table 20 shows the remaining elements of the NuSMV encoding of an HCPN model that contribute to the state space of the Transition System.

Component of NuSVM Encoding	NuSMV variable	Corresponding NuSMV variable domain size	State Space Contribution (in bits), n := number of handlers of substack k := number of times the corresponding HCPN module instance is used in the HCPN model	
			without optimizations	with optimization
Depth Counter	ts_depth	n	$\lceil log_2 n \rceil$	$\lceil log_2 n \rceil$
Transition Relation helper	firedTransition	$\lvert T \rvert$	$\lceil log_2(\lvert T \rvert) \rceil$	$\lceil log_2(\lvert T \rvert) \rceil$

Table 20: State Space Minimization - non-HCPN state space contributors

The Depth Counter

The Depth Counter is a component that we introduce into the NuSMV encoding of the corresponding HCPN model. The Depth Counter functionality is implicit in the HCPN model. The function of the Depth Counter is to keep track of the current execution depth

of the PAM Stack Execution. The HCPN model does not have an explicit notion of "depth", but structurally, the HCPN model implicitly specifies the "depth" of each HCPN module HANDLER_<x> structure via the placement of HANDLER_<x> module instance in relation to other HANDLER_<x> module instances. Specifically, given a HANDLER_<x> module instance, where $x \in 0..n-1$, where n is the number of handlers comprising the PAM Stack, then the depth of HANDLER_<x> is x, i.e. HANDLER_0, is the first handler, and the depth of this handler is 0.

The Depth Counter is a critical factor in state space minimization. It is the Depth Counter that allows us to re-use the NuSMV variables associated with HCPN module ACTION instances. This way, as shown above, the number of HCPN module ACTION instances is "constant" ($0\ or\ 1$) in the number of k times ($0, if\ k = 0, 1, if\ k > 0$) that the HCPN module ACTION instance is used in the HCPN model, as opposed to linear in k.

Specifically, each time we transition into a new depth, i.e. via the firing of transition tInit_1, or via the firing of HCPN module Action transitions tD_<x>_<k_x>_0, or tD_<x>_<k_x>_1 (if exists, i.e. ACTION_DONE), then we update the Depth Counter to hold the value of the depth of the handler which is going to be executed next.

Additionally, only for the transitions tD_<x>_<k_x>_0 and tD_<x>_<k_x>_1, in a sense, we add a "transition guard" to these transitions. This "transition guard" only exists in the NuSMV encoding. This "transition guard" does not exist in the HCPN model encoding. The template for the generation of this "transition guard" is: [d = <x>], where x is the handler depth of the transition, i.e. given tD_1_0_0, the transition of

ACTION_OK_1_0 HCPN module instance in the HCPN model of ACME Corp (Figure 62 on page 96), the "transition guard" generated is: [d = 1]. This means that in the Transition System, the NuSMV encoding representing the firing of the transition tD_1_0_0 will not fire the transition if the value of the "token" d is not 1.

The "token" d is taken as being the current depth. The value of d is taken to be the current value of the Depth Counter, i.e. the value of the NuSMV variable ts_depth.

The above concepts are illustrated in Figure 72. This NuSMV Depth Counter can be interpreted as an HCPN Fusion Set ts_depth, where two edges are added, one outgoing from ts_depth, to the transition in question, and one incoming to ts_depth, from the transition in question. The guard is also assigned to the transition in question. Then, the transition becomes enabled, only if the current value of ts_depth equals the depth of the handler to which the transition belongs to.

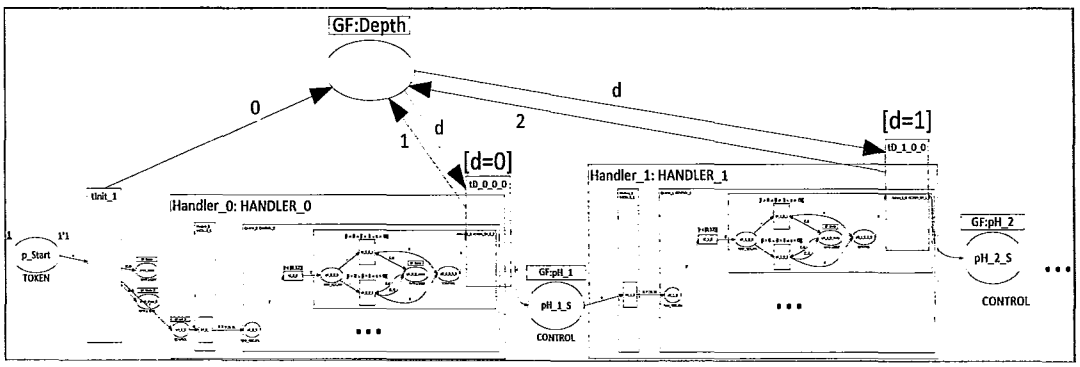

Figure 72: NuSMV - the Depth Counter

Semantically, the Depth Counter ensures that even if we have a single set of NuSMV variables representing multiple HCPN module ACTION_<name(x)>_<x> _<k_x> instances at multiple depths (each Action instance must belong to a distinct depth, by

definition), then only one transition tD_<x>_<k_x>_0 or tD_<x>_<k_x>_1 will be enabled. This ensures that, semantically, the next handler chosen to be executed is chosen by the appropriate transition. For example, in Figure 72, only one of tD_0_0_0 and tD_1_0_0 can become enabled and fire, consequently, choosing the next handler to be executed. Summarizing, the Depth Counter allows us to use one set of NuSMV variables to encode multiple HCPN module ACTION instances. This allows us to achieve constant state space contribution in the number of handlers n and Action module instances k (in contrast to linear contribution).

We discuss the formulation of the Transition Relation, including the firedTransition helper below.

Human Readability and Transition System-HCPN Similarity

The naming conventions beween elements of the HCPN, places and transitions, and their NuSMV counterparts, variables and transition relation terms, respectively, were kept same, where possible, and similar otherwise. Also, the NuSMV encoding reflects the HCPN net structure, i.e. relation between places and transitions, with a similar relationship between NuSMV variables, and the Transition Relation that acts on these variables.

For example, given an HCPN place called p_End, we define the corresponding NuSMV variable called p_End.

For example, when defining a transition relation, we make use of naming conventions such as 'firedTransition', 'guardEnabled', etc.

As a result, when one is working with the Transition System, i.e. model creation, formula specification, model checking, and model checking, it is straight forward to interpret and to relate NuSMV concepts back to the corresponding HCPN and its PAM Stack Instance.

NuSMV Encoding Implementation

As mentioned above, generation of a NuSMV encoding of a Transition System is done automatically by pamtester-fm using hard-coded templates. Below, we omit the details of the format of these templates. Instead, we proceed by example. For all examples in this section, unless noted otherwise, we use the HCPN model instance for ACME Corp (Section Example of "Partial" Unfolding of HCPN Model: HCPN model for ACME Corp on page 91).

NuSMV Modules

An HCPN model is encoded as a single NuSMV module.

Encoding HCPN places as NuSMV variables

An HCPN place not belonging to a Fusion Set corresponds to a distinct NuSMV variable. Hence two distinct HCPN places, both not belonging to a Fusion Set, will each have a distinct NuSMV variable created for them. For example, the HCPN place pC_0_S corresponds to a distinct NuSMV variable also called pC_0_S. Similarly, HCPN place pC_1_S has its own NuSMV variable pC_1_S.

In contrast, given an HCPN Fusion Set, all HCPN places belonging to this Fusion Set are represented by a single NuSMV variable. For example, the Fusion Set GF:Inc contains the following HCPN places: pC_0_Inc, pC_1_Inc, pC_2_Inc, pC_3_Inc and pTerm_Inc. All four of these HCPN places are represented by the NuSMV variable gf_inc.

To specify a NuSMV variable, one must specify three properties of this variable: domain, using the VAR keyword; initial state, using the INIT keyword; and how the value of this variable changes under the Transition Relation of the Transition System, using the TRANS keyword. For example, Table 21 shows the NuSMV encoding of the HCPN place p_Start.

```
VAR p_Start : {EMPTY, 1};
INIT p_Start in { 1 };
TRANS next(p_Start) in
case
  next(firedTransition)=tIinit_1 : EMPTY;
  1 : p_Start;
esac;
```

Table 21: NuSMV encoding of CPN Place `p_Start'

We define a NuSMV variable p_Start, via the VAR keyword, to represent the HCPN place p_Start.

The HCPN place p_Start can either contain a single CONTROL token, or contains no tokens. We encode this by representing the CONTROL token as the value 1, and the place containing no tokens as the value EMPTY. The domain of p_Start is defined to only contain these two possibilities, either p_Start contains no tokens, or it contains the single control token, i.e. {EMPTY, 1}. We encode this in NuSMV syntax by the statement VAR p_Start : {EMPTY, 1}.

The initial marking of the HCPN has the place p_Start contain a single CONTROL token, i.e. 1'CONTROL. We define this in NuSMV by encoding the value of the NuSMV variable p_Start, in the initial state to have value 1. This is done using the INIT keyword, with the statement: INIT p_Start in { 1 }.

Lastly, in the HCPN, if the transition tInit_1 fires, then a CONTROL token is removed from the place p_Start. Otherwise, if the transition tInit_1 is not fired, then the value of the place p_Start stays the same. In our NuSMV encoding, this is encoded as part of the Transition System's Transition Relation, using the TRANS keyword, using the statement TRANS next(p_Start) in case next(firedTransition)=tInit_1 : EMPTY; 1 : p_Start; esac;. The definition of the Transition Relation, and its connection with the CPN's Firing Rule is elaborated on below.

As another example, Table 22 shows the NuSMV encoding of the CPN ending place `p_End`.

```
VAR p_End : {EMPTY, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31 };
INIT p_End in { EMPTY };
TRANS next(p_End) in
case
   next(firedTransition) = tTerm1 : 6;
   next(firedTransition) = tTerm2 : gf_state_stat;
   next(firedTransition) = tTerm3 : 31;
   1 : p_End;
esac;
```

Table 22: NuSMV encoding of CPN Place 'p_End'

Encoding HCPN Fusion Sets as NuSMV variables

Each Global HCPN Place (a set of HCPN places all acting as a single place, but requiring multiple HCPN place instances) was implemented as a single NuSMV variable. This was a critical factor as an arbitrary PAM Stack Instance is near constant in the amount Fusion Set HCPN places, and hence has a constant contribution to the state space of the resulting NuSMV Transition System (see Table 19 on page 59).

Encoding Multi-Coloured HCPN Places as NuSMV Variables

Given an HCPN place whose colour set is a cross product of multiple sets, our NuSMV encoding represents this place via multiple NuSMV variables – one NuSMV variable per colour set. For example, the fusion set GF: state has the following colour set: IMPR x STAT = {EMPTY,-1,0,1} x {EMPTY,0,1,..., 31}. Thus, we define two NuSMV variables, one for the set IMPR and the other for the set STAT.

As an aside, in the case where there is an incoming and outgoing arrow consuming and producing the same multiset of tokens, then this trivial consumption and replacement is not represented by the NuSMV encoding, i.e. transition tA_0_0_1 consumes and places (i,s) from/into pA_0_0_state.

As an aside, in the case that a transition consumes and produces a tokens from/to a place, we do not encode the consumption in NuSMV. We only encode the production. This is because by encoding the production, we implicitly encode the consumption by overwriting the value of the NuSMV variable implementing the HCPN place. For example, tA_0_0_2 consumes (i,s) from pA_0_0_state, and produces (1,r) into pA_0_0_state, but we only encode the producing of (1,r).

An example of a partial encoding is shown in Table 23.

```
VAR gf_state_impr : { EMPTY, -1, 0, 1};
INIT gf_state_impr in {EMPTY};
TRANS next(gf_state_impr) in
case
   --- produce
  next(firedTransition) = tInit_1 : 0;
   next(firedTransition) = tA_0_0_2 : 1;
   next(firedTransition) = tA_0_2_2 : -1;
   next(firedTransition) = tA_0_2_3 : -1;

...

   --- consume
  next(firedTransition) = tTerm_1 : EMPTY;
  next(firedTransition) = tTerm_2 : EMPTY;

   --- otherwise
  1 : gf_state_impr;
esac;
-------------------------------------
VAR gf_state_stat : { EMPTY, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11,
12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28,
29, 30, 31};
INIT gf_state_stat in {EMPTY};
TRANS next(gf_state_stat) in
case
   --- produce
  next(firedTransition) = tInit_6 : 0;
   next(firedTransition) = tA_0_0_2 : 1;
   next(firedTransition) = tA_0_2_2 : pA_0_2_S;
   next(firedTransition) = tA_0_2_3 : 6;

...

   --- consume
  next(firedTransition) = tTerm_1 : EMPTY;
  next(firedTransition) = tTerm_2 : EMPTY;


   --- otherwise

  1 : gf_state_stat;
esac;
```

Table 23: Encoding multi-coloured HCPN places

Encoding HCPN Transitions, Firing Rule as NuSMV Transition Relation

HCPN places are represented as a set of NuSMV variables (1 variable per colour set component). The domains of the HCPN Places are encoded as domains of the NuSMV variables. A marking of the HCPN corresponds to an assignment of values to each of the NuSMV variables. When the HCPN's marking changes to a new marking, this corresponds to a new assignment of values to each of the NuSMV variables.

In HCPNs, firing of transitions changes the marking. In a Transition System, it is the transition relation that changes the values of NuSMV variables.

Hence, we represent the firing of HCPN transitions via the Transition Relation of the Transition System.

We illustrate the NuSMV encoding of a transition relation with a "toy" example. The HCPN is shown in Figure 73.

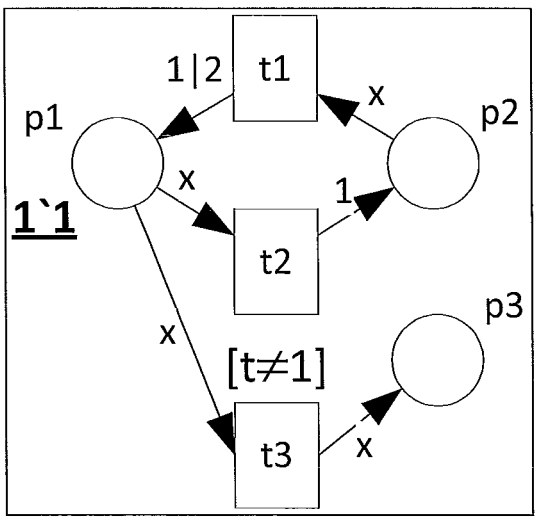

Figure 73: Toy Example - HCPN

The corresponding HCPN "behaviour" is shown in Figure 74.

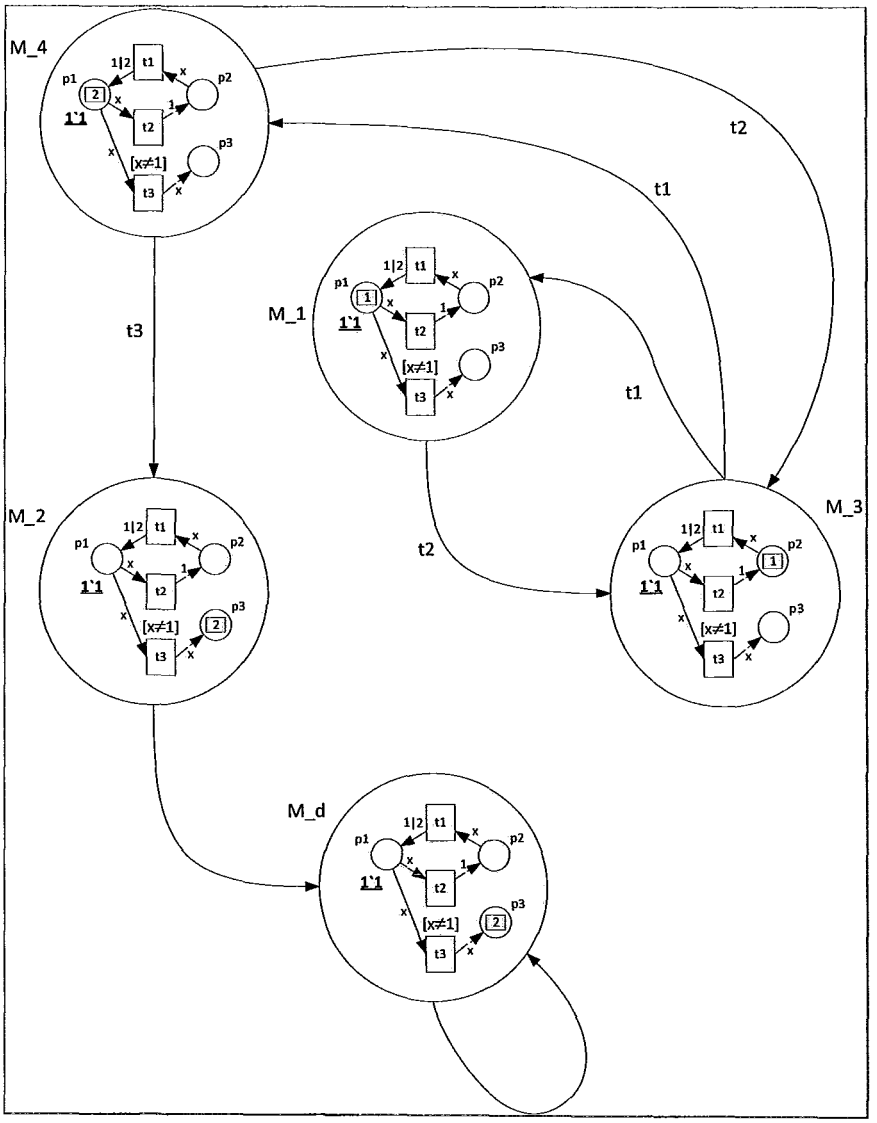

Figure 74: Toy Example - HCPN "behaviour"

In this case, the initial marking of this HCPN is M_1, then the corresponding NuSMV encoding is as follows:

```
--- HCPN place encoding
--- declare variables, and define domain of variables
VAR p1 : { EMPTY, 1, 2 };
VAR p2 : { EMPTY, 1, 2 };
VAR p3 : { EMPTY, 1, 2 };

--- initialize variables
INIT p1 in { 1 };
INIT p2 in { EMPTY };
INIT p3 in { EMPTY };

--- specify how the HCPN marking changes under a firing of a
transition
-- HCPN place p1
TRANS next(p1) in
case
     next(firedTransition) = t1 : {1,2}; -- choice, either 1 or
     next(firedTransition) = t2 : EMPTY; -- consume token
     next(firedTransition) = t3 : EMPTY; -- consume token
     1 : p1; -- default, keep value same
esac;

-- HCPN place p2
TRANS next(p2) in
case
     next(firedTransition) = t2 : 1;
     next(firedTransition) = t1 : EMPTY;
     1 : p2;
esac;

-- HCPN place p3
TRANS next(p3) in
case
     next(firedTransition) = t3 : p1;
     1 : p3;
esac;

--- HCPN Firing Rule encoding
--- declare the firedTransition variable Transition Relation helper
VAR firedTransition : { DNE, t1, t2, t3 };

--- initialize firedTransition
INIT firedTransition in { DNE };

--- define the HCPN firing rule
-- first, define what it means in NuSMV for a transition to be
enabled
- we need the preset of our transitions to be populated with the
required tokens for consumption
```

```
DEFINE pre_t1 := p2 != EMPTY;
DEFINE pre_t2 := p1 != EMPTY;
DEFINE pre_t3 := p1 != EMPTY;

-- second, define guards
- we need the guard of our transitions to evaluate to TRUE
DEFINE guard_t1 := TRUE;
DEFINE guard_t2 := TRUE;
DEFINE guard_t3 := p1 != 1;

-- third, combine preset and guard conditions into enabled condition
- we need the enabled condition  to require that both the preset and
guard conditions are TRUE
DEFINE t1_enabled := ( pre_t1 & guard_t1 );
DEFINE t2_enabled := ( pre_t2 & guard_t2 );
DEFINE t3_enabled := ( pre_t3 & guard_t3 );

-- fourth, complete the transition relation definition
TRANS ( t1_enabled & next(firedTransition) = t1 )
     |( t2_enabled & next(firedTransition) = t2 )
     |( t3_enabled & next(firedTransition) = t3 )
     |( !t1_enabled & !t2_enabled & !t3_enabled ) &
next(firedTransition) = DNE)
```

Figure 75: Toy example - HCPN and corresponding NuSMV Encoding including Transition System encoding

The transition relation definition for the variable firedTransition ensures that the firedTransition is set to one of t1, t2, t3 or DNE. This is because it always evaluates to TRUE since, after simplification, it is of the form $(\neg\phi \vee \phi)$:

```
(p2 != EMPTY) | (p1 != EMPTY) | (p1 != EMPTY & p1 != 1)
|
(!( p2 != EMPTY) & !(p1 != EMPTY) & !(p1 != EMPTY & p1 != 1))
```

Table 24: Transition Relation - firedTransition NuSVM variable

Thus, the transition relation function ->, in our case the function next(), under all circumstances is defined (total) for the variable firedTransition, i.e. for all states, the variable firedTransition has a definition for its next value.

In turn, this fact causes the rest of the system variables to be set, due to their portion of the Transition Relation definition. The number 1, in the case definitions "1 : p1", "1 : p2", "1 : p3", is the default case. The default case always evaluates to true, and is always chosen if there are no other cases that evaluate to true. Since all TRANS constraints, for all of the variables, contain a default "1: ...;" entry, hence all variable case statements also evaluate to true under all conditions.

```
TRANS next(p1) in
case
     next(firedTransition) = t1 : {1,2}; -- choice, either 1 or 2
     next(firedTransition) = t2 : EMPTY; -- consume token
     next(firedTransition) = t3 : EMPTY; -- consume token
     1 : p1; -- default, keep value same
esac;
TRANS next(p2) in
case
     next(firedTransition) = t2 : 1;
     next(firedTransition) = t1 : EMPTY;
     1 : p2;
esac;
TRANS next(p3) in
case
     next(firedTransition) = t3 : p1;
     1 : p3;
esac;
```

Table 25: Transition Relation - HCPN Places

Thus, since all system variables have a defined value in the next state, i.e. via the next() transition function in their respective TRANS constraint section, thus, the Transition System as a whole has a defined next state, under all conditions. Thus, the transition relation is defined for all conditions. Thus, TRANS is total - a requirement for Transition System transition relations (recall that for all states s of the system, there must exist a state s' of the system such as that s -> s', where -> is the transition relation. In this case, this requirement holds.

Lastly, note that although the HCPN marking M_2 is a deadlock marking, i.e. the HCPN cannot reach a new marking from M_2, the corresponding Transition System still moves to another state! Precisely, when M_2 is reached (i.e. the HCPN enters a deadlock marking), the next() function sets firedTransition = t3, and thus causes all system variables p1, p2, and p3 to retain old values, i.e. next(p1)=p1, next(p2)=p2, next(p3)=p3. This Transition System variable valuation is what defines the Transition System state. In other words, the Transition System state is viewed as a 4-tuple containing the values of the Transition System variables. Mainly, M_2 is described as (firedTransition, p1, p2, p3)=(t3, EMPTY, EMPTY, 2).

Now, as mentioned above, note that although the underlying HCPN does not change its marking anymore, the Transition System still moves into another state! Once in Transition System state M_2, the TRANS constraint causes the following updates to the system variables: next(firedTransition) = DNE, next(p1)=p1, next(p2)=p2, next(p3)=p3, thus the new state is described by (firedTransition, p1, p2, p3)=(DNE, EMPTY, EMPTY, 2), and thus, is not the state M_2. We define a new state, called M_d, i.e. M_d := (DNE, EMPTY, EMPTY, 2). Hence, TRANS causes the Transition System to change state from M_2 to M_d.

This is illustrated in Figure 76.

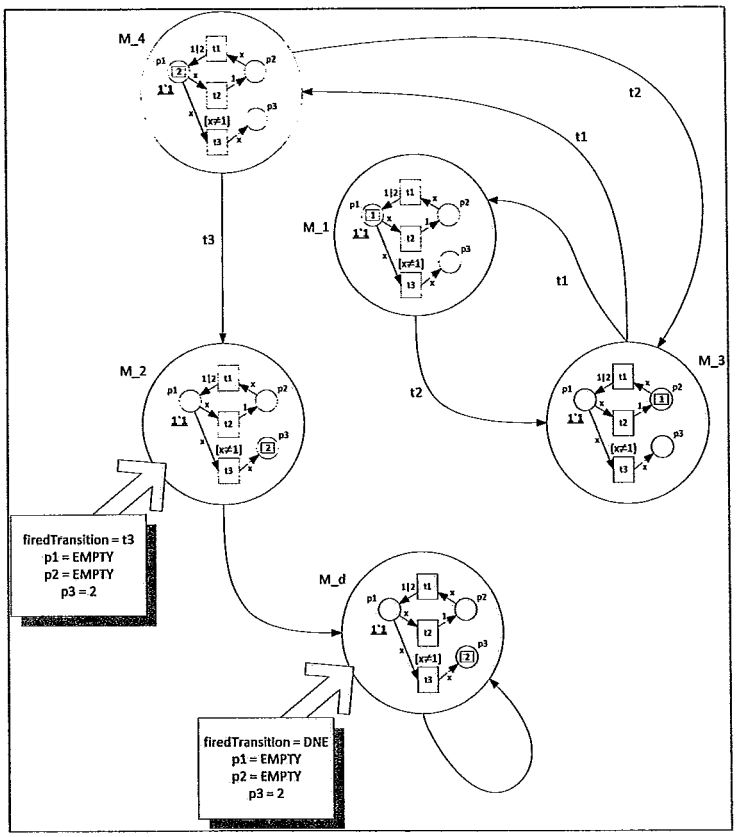

Figure 76: The System variables underlying the Transition System

Specifying and Verifying "Security Properties" of Linux-PAM Configurations

Introduction

Given the generated Transition System, which describes the "behaviour" of an HCPN model of a PAM Stack Instance execution, we can analyze the behaviour of this HCPN. This analysis simulates the effect of the HCPN Firing Rule on the markings of the HCPN.

We do this by simulating the possible state changes of the corresponding Transition System.

So, first, we use NuSMV to build a Transition System model instance, and then we ask NuSMV to simulate the possible state changes of this Transition System instance. Furthermore, we request that NuSMV, while iterating through these state changes, checks whether or not certain system properties (atomic expressions in Atoms) hold on the iterated-over system states (use the labelling function L to check whether or not a system property critical to the security of the corresponding Linux-PAM Configuration, aka a "security property" ϕ is tre in the iterated-over state s, i.e. is it the case that $\phi \in L(s)$?

Determining All Possible PAM Stack Execution Return Values

For example, suppose we are considering the following Transition System state, shown in Figure 77.

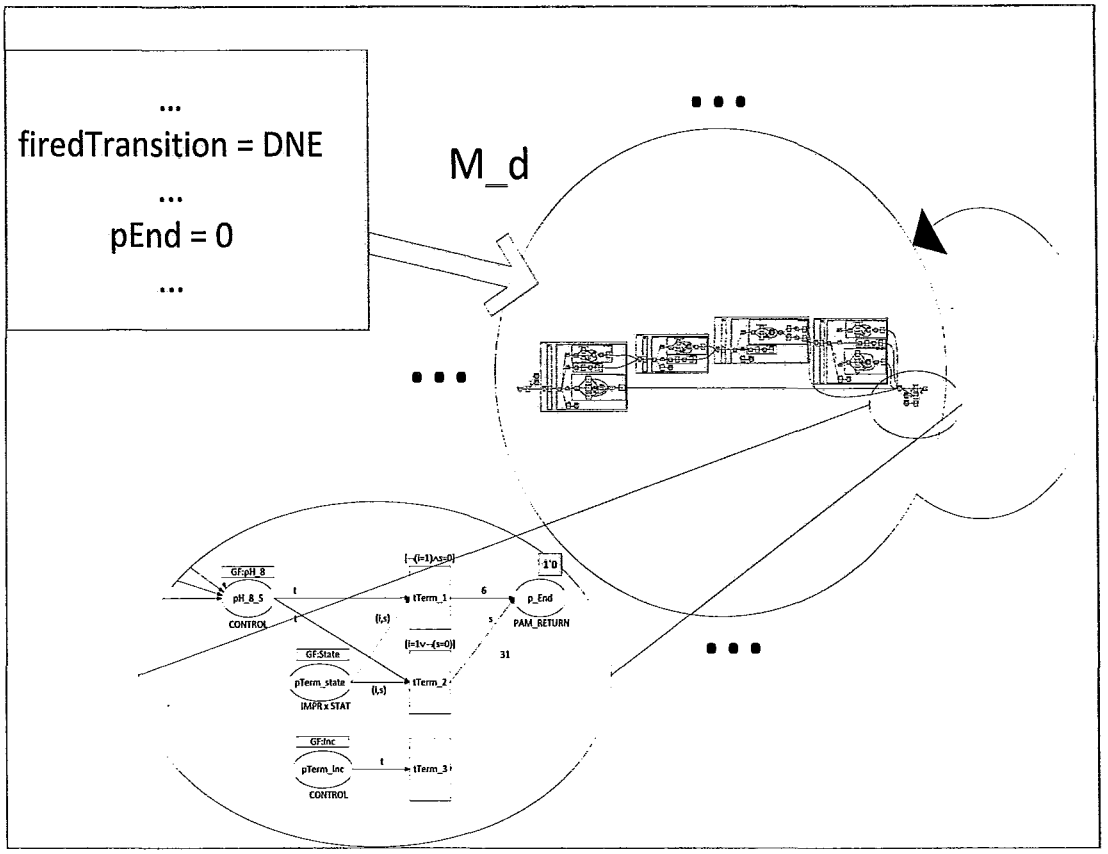

Figure 77: Labeling Function - Example - ACME Corp

Here, as discussed in prior sections, the underlying HCPN is in a deadlock state. The corresponding Transition System state is also in a "deadlock" state in a sense that the Transition System is not able to change from the state M_d to another (distinct from M_d) state. This is because the underlying HCPN Marking cannot reach any other marking. This situation corresponds to the end of the PAM Stack Execution of the corresponding PAM Stack Instance of the HCPN model.

We are interested in this particular situation because we want to know the value of the NuSMV variable pEnd. The value of NuSMV variable pEnd corresponds to the value of

the PAM_RETURN token value residing in the HCPN place p_End (if exists, otherwise corresponds to EMPTY) in the underlying HCPN. The PAM_RETURN value contained in p_End corresponds to the PAM Stack Execution PAM_RETURN value returned by Linux-PAM at the end of the PAM Stack Execution. Semantically, the resulting PAM_RETURN value of the PAM Stack Execution determines whether or not the authentication-related functionality was successful, i.e. the user was able to authenticate, the user account check was successful, the user password update was successful, the log on session set up or tear down was successful, the setting of user credentials was successful.

For example, if p_End = 0, then the PAM_RETURN value is PAM_SUCCESS = 0. This means that the authentication-related functionality (i.e. the client-called Management Function f) was a success. On the other hand, if p_End belongs to {1, 2, ... , 30}, then the authentication-related functionality was not successful, i.e. the user failed authentication, the user account check failed, the user password update failed, the log on session set up failed, the log on session tear down failed, the setting of user credentials failed. Lastly, if p_End = 31, then a "stacked" PAM requested that the PAM Stack Execution is paused so that the user can provide additional information for the proper carrying out of the authentication-related functionality.

When a Linux-PAM Adminsitrator creates a Linux-PAM Configuration, one of the first questions that arise is: what are the possible PAM_RETURN values of this Linux-PAM Configuration.

At this time, the methods for determining the set of possible PAM_RETURN values, for a Linux-PAM configuration, are limited.

For example, there exists a software tool called pamtester. The administrator tells the pamtester tool which Linux-PAM service, for which Management Function to test. In turn, pamtester establishes an authentication process with Linux-PAM (via pam_start() Client API function), and calls the Management Function in question. The obtained PAM_RETURN is then reported back to the user. The limitation here is that the PAM_RETURN result used a single PAM Stack Execution trace to obtain this result. There was no enumeration of possible PAM Stack Execution traces.

Furthermore, the Linux-PAM administrator may attempt to "manually" enumerate the possible PAM Stack Execution traces. This method is based on manual enumeration, including changing environment settings, i.e. provide a password during authentication, don't provide a password during authentication. This method is based on Linux-PAM Administrator experience and knowledge of Linux-PAM. Hence, this method is also limited.

The next section describes how pamtester-fm enumerates all possible PAM Stack Execution traces, and hence provides all possible PAM_RETURN values for an arbitrary Linux-PAM Configuration, for an arbitrary Linux-PAM Service, for an arbitrary Management Function.

Formal Specification of "Security Properties" via CTL

Pamtester-fm, in its current state tackles the problem of enumerating all possible PAM Stack Execution return values. To do this, pamtester-fm model checks Transition System describing all of the possible PAM Stack Executions of the corresponding Linux-PAM Instance, generated from the Linux-PAM Configuration of interest.

This model checking consists of checking, for all PAM_RETURN values, if it is true that for all computation paths starting at the intial state, for all states comprising each computation path, the value of the variable p_End does not equal the PAM_RETURN value. Precisely, the formal specification is show in Figure 78.

```
set verbose_level 0
unset counter_examples
check_ctlspec -p "AG pend != 0"
check_ctlspec -p "AG pend != 1"
check_ctlspec -p "AG pend != 2"
check_ctlspec -p "AG pend != 3"
check_ctlspec -p "AG pend != 4"
check_ctlspec -p "AG pend != 5"
check_ctlspec -p "AG pend != 6"
check_ctlspec -p "AG pend != 7"
check_ctlspec -p "AG pend != 8"
check_ctlspec -p "AG pend != 9"
check_ctlspec -p "AG pend != 10"
check_ctlspec -p "AG pend != 11"
check_ctlspec -p "AG pend != 12"
check_ctlspec -p "AG pend != 13"
check_ctlspec -p "AG pend != 14"
check_ctlspec -p "AG pend != 15"
check_ctlspec -p "AG pend != 16"
check_ctlspec -p "AG pend != 17"
check_ctlspec -p "AG pend != 18"
check_ctlspec -p "AG pend != 19"
check_ctlspec -p "AG pend != 20"
check_ctlspec -p "AG pend != 21"
check_ctlspec -p "AG pend != 22"
check_ctlspec -p "AG pend != 23"
check_ctlspec -p "AG pend != 24"
check_ctlspec -p "AG pend != 25"
check_ctlspec -p "AG pend != 26"
check_ctlspec -p "AG pend != 27"
check_ctlspec -p "AG pend != 28"
check_ctlspec -p "AG pend != 29"
check_ctlspec -p "AG pend != 30"
check_ctlspec -p "AG pend != 31"
```

Figure 78: Computation Tree Logic formula list for obtaining the set of all possible PAM_RETURN values of a PAM Stack Instance Execution

Formal Verification of "Security Properties" via Model Checking

Pamtester-fm instructs NuSMV to build the transition system model, and then to model check this model.

Pamtester-fm uses the command line syntax in Figure 79 to instruct NuSMV to do this.

```
nusmv -v 0 -dynamic -load ${nusmv_test_profile} ${nusmv_model_file}
```

Figure 79: Pamtester-rm NuSMV Command Line Syntax for Checking of Linux-PAM Configurations

The algorithm used for the model checking of the system is critical. The –dynamic NuSMV option specifies that NuSMV uses "dynamic ordering of BDD variables". When we did not use the '-dynamic' option, we observed that even simple Linux-PAM configurations (ones reaching five or six "stacked" PAMs) exhausted the memory on our test system[6]. Also, time-wise, model-checking times were reaching close to 8 hours before computer memory was exhausted. In contrast, when using the '-dynamic option', we did not reach such limitations at five or six "stacked" PAMs. In fact, our HCPN models of PAM Stack Instances reaching 10s and 100s of "stacked" PAMs were handled relatively "easily", i.e. 1s to10s of seconds while using under 100 MB of RAM.

The '-load ${nusmv_test_profile}' arguments instruct NuSMV to carry out extra commands, as specified in the file ${nusmv_test_profile}. The contents of this file are shown in Figure 80.

[6] PC with 3 GB RAM and a Dual Core Intel 2 Ghz CPU.

```
set on_failure_script_quits
set verbose_level 0
echo
"
_____________________________________________________________________________
_____________________________________________________________________________"
echo ">>>>>> NuSMV: building a NuSMV-internal model for the NuSMV-
encoded hierarchical Coloured Petri Net representing the Linux-PAM
configuration:"
go;
time;
echo
"
_____________________________________________________________________________
_____________________________________________________________________________"
echo ">>>>>> NuSMV: running the verification of security properties on
the NuSMV-internal model:"
echo
"
_____________________________________________________________________________
_____________________________________________________________________________"
source test.check_all_return_values;
echo
"
_____________________________________________________________________________
_____________________________________________________________________________"
time;
echo
"
_____________________________________________________________________________
_____________________________________________________________________________"
quit;
```

Figure 80: Pamtester-fm NuSMV Test Profile script

The test profile script opens another file called test.check_all_return_values, via the 'source test.check_all_return_values;' command. The contents of this file are the "security properties" to be checked on theTransition System models. The contents of this file were already shown above, in Figure 78 on page 139.

The last argument, '${nusmv_model_file}', identifies a file containing a list of files, each of these files containing a NuSMV Syntax encoding of the Transition Systems describing the PAM Stack Instance Executions to be checked (for whether or not the "security properties" hold during simulation). The contents of this file are shown in Figure 81.

```
../smv/crond_smv.txt
../smv/cups_smv.txt
../smv/gdm-autologin_smv.txt
../smv/gdm_smv.txt
../smv/login_smv.txt
../smv/passwd_smv.txt
../smv/smtp_smv.txt
../smv/sshd_smv.txt
../smv/sudo_smv.txt
../smv/su_smv.txt
../smv/wireshark_smv.txt
```

Figure 81: Pamtester-fm file list containing NuSMV encodings of PAM Stack Instance Execution possibilities for Linux-PAM Configurations of Production-grade Linux-PAM Services

Once NuSMV loads the security properties from Figure 78 on page 139, then NuSMV uses the Test Profile in Figure 80 on page 141 to carry out automated transition system model building and model checking, where the transition system encodings are obtained from the files listed in Figure 81.

The Transition System encodings in Figure 81 represent production-grade Linux-PAM configurations take from Red Hat Fedora Core 6.

To illustrate, an example of the output from an automated pamtester-fm session was captured and a portion of it is presented in Figure 82. This output shows the automated verification of "security properties" of Linux-PAM Configurations for the crond and cupsd Linux-PAM Clients.

```
[kkulbakas@localhost verification]$ source test_driver.sh update2.nusmv_test_profile
update2.nusmv_model_list


\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
****** This is pamtester-fm.
****** Test Harness Driver, v.0.0.1 --EXECUTION START on Mon Jul 20 20:37:22 EDT 2009.


<nusmv_test_profile> : update2.nusmv_test_profile
<nusmv_model_list_file>: update2.nusmv_model_list


\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
\\\\\\\\\\\\\\\\\\\\
****** This is pamtester-fm.
****** Invoking test number '1' out of '11' on Mon Jul 20 20:37:22 EDT 2009


<nusmv_model_file> : ../smv/crond_smv.txt
<nusmv_test_profile> : update2.nusmv_test_profile
NuSMV command line : nusmv -v 0 -dynamic -load update2.nusmv_test_profile -i ../smv/crond_smv.txt


\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
\\\\\\\\\\\\\\\\\\\\
FILE ->>> update2.nusmv_test_profile
*** This is NuSMV 2.4.3 (compiled on Sat Apr 4 20:10:25 UTC 2009)
*** For more information on NuSMV see <http://nusmv.irst.itc.it>
*** or email to <nusmv-users@irst.itc.it>.
*** Please report bugs to <nusmv@irst.itc.it>.


>>>>>> NuSMV: building a NuSMV-internal model for the NuSMV-encoded hierarchical Coloured Petri Net representing the
Linux-PAM configuration:
elapse: 2.3 seconds, total: 2.3 seconds


>>>>>> NuSMV: running the verification of security properties on the NuSMV-internal model:


--specification AG pend != 0 is false
--specification AG pend != 1 is true
--specification AG pend != 2 is true
--specification AG pend != 3 is false
--specification AG pend != 4 is true
--specification AG pend != 5 is false
--specification AG pend != 6 is true
--specification AG pend != 7 is false
--specification AG pend != 8 is true
--specification AG pend != 9 is true
--specification AG pend != 10 is true
--specification AG pend != 11 is true
--specification AG pend != 12 is true
--specification AG pend != 13 is true
--specification AG pend != 14 is true
--specification AG pend != 15 is true
--specification AG pend != 16 is true
--specification AG pend != 17 is true
--specification AG pend != 18 is true
--specification AG pend != 19 is true
--specification AG pend != 20 is true
--specification AG pend != 21 is true
--specification AG pend != 22 is true
--specification AG pend != 23 is true
--specification AG pend != 24 is true
--specification AG pend != 25 is true
--specification AG pend != 26 is false
--specification AG pend != 27 is true
--specification AG pend != 28 is true
--specification AG pend != 29 is true
--specification AG pend != 30 is true
--specification AG pend != 31 is true


elapse: 11.3 seconds, total: 13.5 seconds


Quitting the BMC package...
Done
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
\\\\\\\\\\\\\\\\\\\\
****** This is pamtester-fm.
****** Invoking test number '2' out of '11' on Mon Jul 20 20:37:40 EDT 2009
```

```
<nusmv_model_file> : ../smv/cups_smv.txt
<nusmv_test_profile> : update2.nusmv_test_profile
NuSMV command line : nusmv -v 0 -dynamic -load update2.nusmv_test_profile -i ../smv/cups_smv.txt


\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
\\\\\\\\\\\\\\\\\\\\
FILE ->>> update2.nusmv_test_profile
*** This is NuSMV 2.4.3 (compiled on Sat Apr 4 20:10:25 UTC 2009)
*** For more information on NuSMV see <http://nusmv.irst.itc.it>
*** or email to <nusmv-users@irst.itc.it>.
*** Please report bugs to <nusmv@irst.itc.it>.


>>>>>> NuSMV: building a NuSMV-internal model for the NuSMV-encoded hierarchical Coloured Petri Net representing the
Linux-PAM configuration:
elapse: 1.9 seconds, total: 1.9 seconds


>>>>>> NuSMV: running the verification of security properties on the NuSMV-internal model:


--specification AG pend != 0 is false
--specification AG pend != 1 is true
--specification AG pend != 2 is true
--specification AG pend != 3 is false
--specification AG pend != 4 is true
--specification AG pend != 5 is false
--specification AG pend != 6 is true
--specification AG pend != 7 is false
--specification AG pend != 8 is true
--specification AG pend != 9 is true
--specification AG pend != 10 is true
--specification AG pend != 11 is true
--specification AG pend != 12 is true
--specification AG pend != 13 is true
--specification AG pend != 14 is true
--specification AG pend != 15 is true
--specification AG pend != 16 is true
--specification AG pend != 17 is true
--specification AG pend != 18 is true
--specification AG pend != 19 is true
--specification AG pend != 20 is true
--specification AG pend != 21 is true
--specification AG pend != 22 is true
--specification AG pend != 23 is true
--specification AG pend != 24 is true
--specification AG pend != 25 is true
--specification AG pend != 26 is false
--specification AG pend != 27 is true
--specification AG pend != 28 is true
--specification AG pend != 29 is true
--specification AG pend != 30 is true
--specification AG pend != 31 is true


elapse: 5.1 seconds, total: 7.0 seconds


Quitting the BMC package...
Done
```

Figure 82: Pamtester-fm automated, formal verification of "security properties" of Linux-PAM Configurations

Interpreting NuSMV Model Checking Results

Pamtester-fm utilizes the NuSMV program to build and model check transition system models. Pamtester-fm does this in an automated manner, as shown above. Pamtester-fm obtains the output from NuSMV,and parses, processes and presents the results of this

model checking, as necessary. Continuing the example from above, the output provided by pamtester-fm for the verification of security properties of the Linux-PAM Configuration for user authentication for the login program is shown in Figure 83.

```
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
\\\\\\\\\\\\\\\\\\\
****** This is pamtester-fm.
****** Invoking test number '5' out of '11' on Mon Jul 20 20:38:00 EDT 2009
<nusmv_model_file> : ../smv/login_smv.txt
<nusmv_test_profile> : update2.nusmv_test_profile
NuSMV command line : nusmv -v 0 -dynamic -load update2.nusmv_test_profile -i ../smv/login_smv.txt
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
\\\\\\\\\\\\\\\\\\\
FILE ->>> update2.nusmv_test_profile
*** This is NuSMV 2.4.3 (compiled on Sat Apr 4 20:10:25 UTC 2009)
*** For more information on NuSMV see <http://nusmv.irst.itc.it>
*** or email to <nusmv-users@irst.itc.it>.
*** Please report bugs to <nusmv@irst.itc.it>.
________________________________________________________________________________________________________________________

>>>>>> NuSMV: building a NuSMV-internal model for the NuSMV-encoded hierarchical Coloured Petri Net representing the
Linux-PAM configuration:
elapse: 2.3 seconds, total: 2.3 seconds
________________________________________________________________________________________________________________________

>>>>>> NuSMV: running the verification of security properties on the NuSMV-internal model:
________________________________________________________________________________________________________________________

-- specification AG pend != 0 is false
-- specification AG pend != 1 is true
-- specification AG pend != 2 is true
-- specification AG pend != 3 is false
-- specification AG pend != 4 is true
-- specification AG pend != 5 is false
-- specification AG pend != 6 is true
-- specification AG pend != 7 is false
-- specification AG pend != 8 is true
-- specification AG pend != 9 is true
-- specification AG pend != 10 is true
-- specification AG pend != 11 is true
-- specification AG pend != 12 is true
-- specification AG pend != 13 is true
-- specification AG pend != 14 is true
-- specification AG pend != 15 is true
-- specification AG pend != 16 is true
-- specification AG pend != 17 is true
-- specification AG pend != 18 is true
-- specification AG pend != 19 is true
-- specification AG pend != 20 is true
-- specification AG pend != 21 is true
-- specification AG pend != 22 is true
-- specification AG pend != 23 is true
-- specification AG pend != 24 is true
-- specification AG pend != 25 is true
-- specification AG pend != 26 is false
-- specification AG pend != 27 is true
-- specification AG pend != 28 is true
-- specification AG pend != 29 is true
-- specification AG pend != 30 is true
-- specification AG pend != 31 is false
________________________________________________________________________________________________________________________

elapse: 2.0 seconds, total: 4.3 seconds
________________________________________________________________________________________________________________________

Quitting the BMC package...
Done
```

Figure 83: Pamtester-fm formal verification of 'security properties' for user authentication for the login program

As shown in Figure 83, it took 2.3 seconds for NuSMV to build the transition system model instance. It took another 2 seconds for NuSMV to model check this model instance to determine the truth or falsity of the provided "security properties". The total time for model instance build and model check was 4.3 seconds.

The results of the verification are shown in Figure 84.

```
-- specification AG pend != 0 is false
-- specification AG pend != 1 is true
-- specification AG pend != 2 is true
-- specification AG pend != 3 is false
-- specification AG pend != 4 is true
-- specification AG pend != 5 is false
-- specification AG pend != 6 is true
-- specification AG pend != 7 is false
-- specification AG pend != 8 is true
-- specification AG pend != 9 is true
-- specification AG pend != 10 is true
-- specification AG pend != 11 is true
-- specification AG pend != 12 is true
-- specification AG pend != 13 is true
-- specification AG pend != 14 is true
-- specification AG pend != 15 is true
-- specification AG pend != 16 is true
-- specification AG pend != 17 is true
-- specification AG pend != 18 is true
-- specification AG pend != 19 is true
-- specification AG pend != 20 is true
-- specification AG pend != 21 is true
-- specification AG pend != 22 is true
-- specification AG pend != 23 is true
-- specification AG pend != 24 is true
-- specification AG pend != 25 is true
-- specification AG pend != 26 is false
-- specification AG pend != 27 is true
-- specification AG pend != 28 is true
-- specification AG pend != 29 is true
-- specification AG pend != 30 is true
-- specification AG pend != 31 is false
```

Figure 84: Pamtester-fm verification result for the checking of "security properties" for the login program

Each of the output lines shown in Figure 84 corresponds to a distinct "security property" listed in Figure 78 on page 139.

For example, the output line '—specification AG pend != 0 is false' corresponds to the "security property" 'check_ctlspec -p "AG pend != 0'. In this case, this means that the CTL formula "AG pend!=0" is not true in the transition system model instance. This means that in the corresponding HCPN model, it is false that there is never a token whose value is 0, in the HCPN place p_End. In other words, it is true that at some point, there is

a token of value 0 in the HCPN place p_End. This means that in the corresponding PAM Stack Instance's PAM Stack Execution, there exists some PAM Stack Execution which returns a PAM_RETURN value of 0. Since, in this example, the PAM Stack Instance is the Effective PAM Stack Instance for the Management Function pam_authenticate(), thus, ultimately, this result means that it is possible for a user to authenticate successfully using the login program.

To summarize the verification results in Figure 84, the set of PAM_RETURN values that can be possibly returned during user authentication for the Linux-PAM Configuration for the login program is comprised of the following:

- PAM_SUCCESS = 0,
- PAM_ SERVICE_ERR = 3,
- PAM_BUF_ERR = 5,
- PAM_AUTH_ERR = 7,
- PAM_ABORT = 26,
- PAM_INCOMPLETE = 31.

RESULTS

At first, it turned out that even with the structural optimizations done during HCPN specification (i.e. minimizing domains of HCPN places, minimizing amount of HCPN places used, not using any HCPN places for Transitions, etc., see Table 15 on page 110 to Table 20 on page 118), and then, minimizations done during NuSMV encoding of HCPN specifications (i.e. Depth Counter, see Figure 72 on page 120), we still succumbed to the State Space Explosion problem.

To overcome the State Space Explosion problem, we used another model verification algorithm, called dynamic reordering. NuSMV utilizes the CUDD library to use this algorithm. The CUDD library implements many model verification algorithms.

NuSMV does not use the dynamic reordering algorithm by default. One has to explicitly instruct NuSMV to use it. Pamtester-fm does this by using the '-dynamic' command line argument when executing NuSMV (see Figure 79 on page 140).

To illustrate the drastic difference between the usage of the default model verification algorithm used by NuSMV, and the dynamic reordering model verification algorithm, Table 26 on page 153 shows a summary of our test results when using the default NuSMV algorithm, and the Table 27 on page 154 shows a summary of our testing when using the dynamic reordering algorithm.

For these tests, we used a Linux-PAM Configuration that was simple, yet provided some complexity. Specifically, we started with a "base case" Linux-PAM configuration, shown in Figure 85.

```
testservice auth [default=ok] pam_nologin.so
```

Figure 85: State Space Explosion - Test "Base Case" – Linux-PAM Configuration

This base case uses the Linux-PAM Service testservice. The Linux-PAM Service Group is auth. The Complex Control specifies that all PAM_RETURN values, i.e. {0,1, …, 31}, are to be mapped to the Action 'ok' by the Control function, i.e. $(\forall\, return | return \in \{0,1, \ldots, 31\}: C(return) = -1)$ holds. The last token is the PAM: pam_nologin.so PAM. Recall, in our model, we define all PAM as a single HCPN transition which produces exactly one PAM_RETURN value when fired. This return value belongs to the set of the modeled, possible returned values of the pam_nologin.so PAM. In this case, we modeled this set to be the values PAM_SUCCESS = 0, PAM_BUF_ERR = 5, PAM_AUTH_ERR = 7, PAM_UNKNOWN_USER = 10, PAM_IGNORE = 25.

The HCPN model of this base case is shown in Figure 86.

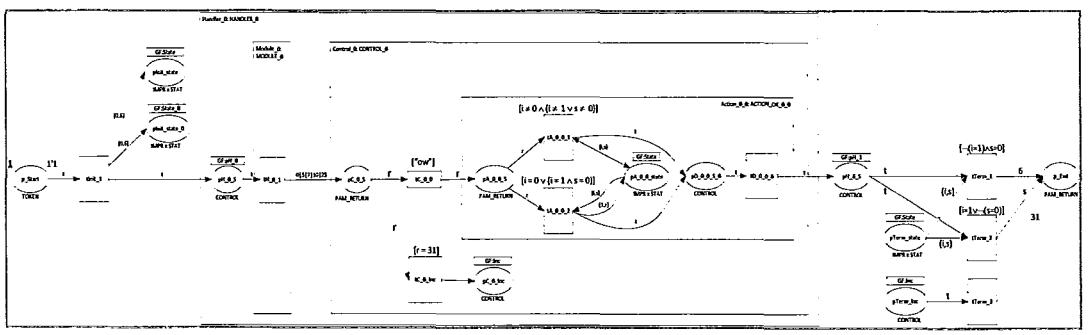

Figure 86: State Space Explosion - Test "Base Case" – HCPN model

Then, we ensued our testing by creating Linux-PAM configurations, increasing by the number of "stacked" PAMs. Each such configuration was created using the base case

configuration from Figure 85. For example, a test Linux-PAM configuration where the number of "stacked" PAMs is nine is shows in Figure 87.

```
testservice auth [default=ok] pam_nologin.so
testservice auth [default=ok] pam_nologin.so
testservice auth [default=ok] pam_nologin.so
testservice auth [default=ok] pam_nologin.so
testservice auth [default=ok] pam_nologin.so
testservice auth [default=ok] pam_nologin.so
testservice auth [default=ok] pam_nologin.so
testservice auth [default=ok] pam_nologin.so
testservice auth [default=ok] pam_nologin.so
```

Figure 87: State Space Explosion - Test "Base Case" – Linux-PAM Configuration with 9 "stacked" PAMs

We started at a configuration with one module, and proceeded to increase the number of modules by one, for each subsequent test case. Our test platform was a machine with the following specification: 2.0 Ghz Dual Core CPU, 3 GB RAM, GNU/Linux. We experienced State Space Explosion, both, in terms of storage requirement of working memory (RAM), as well as time. In terms of both RAM and time, we observed exponential increase in both, as we increased the number of "stacked" PAMs.

During Transition System verification, first, NuSMV builds a model instance of this Transition System. Then, secondly, NuSMV proceeds to model check this model instance. During the model checking stage of model verification, the "security properties" that were used in this model checking where the CTL formulas checking for which PAM_RETURN values the corresponding PAM Stack Execution could possibly return, as shown in Figure 78 on page 139.

Although we saw evidence of State Space Explosion in terms of both RAM and time requirements, it was RAM that was the ultimate "show stopper", occurring at the first stage of verification – the model building stage.

Evidence of the State Space Explosion Problem

When we tested a configuration with five, six and seven "stacked" PAMs, the RAM requirement for the model building were 260, 1162 and 1877 MB of RAM, respectively. The model building stage of the verification, for the configuration with eight "stacked" PAMs, did not finish. Instead, the NuSMV process crashed with an error message "add_to_bdd: result = NULL". Upon investigation, this error was generated by a function supplied by the CUDD[6] library, the function defined in the file C source code file of CUDD, the file named dd.c. Upon further investigation, the CUDD user manual mentions that this error indicates that the process has ran out of memory. This was supported by the fact that our system memory was exhausted by the NuSMV process. The time that it took for the RAM to be exhausted was 233 seconds.

In terms of time and the State Space Explosion, it was observed that for test configurations of three, four, five and six "stacked" PAMs, the model checking stage of the model verification process took 1, 6, 214 and 1539 seconds, also an exponential curve.

Due to these results, it was clear that using the default NuSMV model verification algorithm, it was not realistic for pamtester-fm to verify production Linux-PAM

[6] NuSMV uses the CUDD package for its library which implements model building algorithms.

configurations. This is because, even though, the average observed Linux-PAM Configuration (default Linux-PAM installatioin on Fedora Core 6) is only 5 "stacked" PAMs, none-the-less, our test configuration used only a single action – Action 'ok'. This was a problem because production Linux-PAM Configurations use, on average, between 3 and 4 Actions[8].

The above results are summarized in Table 26 (N/A – not available, N/R – not recorded, comments between < >).

[8] In the production Linux-PAM configurations that were tested, most configuration lines use the Basic Control syntax, which uses between 3 and 4 Actions per Basic Control token.

			Build Model					Model Check model	
# of "stacked" PAMS	# of state variables[8]	Read model, flatten hierarchy, Build flat model (seconds)	Time used (seconds)	RAM used (MB)	(# BDD nodes, in millions) (# clusters)	BDDs/Cluster (Cluster1) (Cluster2) . . . (Cluster6)	Transition Relation Totality Check (seconds)	Time used (seconds)	RAM used (MB)
1	96	0	0.1	N/R	N/R	N/R	0.3	0.0	N/R
2	106	0	0.2	N/R	N/R	N/R	0.9	0.1	N/R
3	116	0	0.5	N/R	N/R	N/R	5.6	0.7	N/R
4	224	0	2.8	N/R	N/R	N/R	28.3	5.7	100
5	132	0	19.6	260	N/R	N/R	187.9	214	315
6	142	0	118.5	1162	14321	2782970	N/R	1539	865
						2224			
						1242			
					6	20023			
						1070			
						411			
7	152	0	664.8	1877	75364	12651772	N/R	N/R	N/R
						4411			
						4049			
					6	4041			
						21775			
						672			
8	160	0	<CRASH (OCCURRED IN BDD PACKAGE)>	<CRASH>	122481	<CRASH>	<CRASH>	<CRASH>	<CRASH>
					<CRASH>				

Table 26: State Space Explosion - Results Summary

Overcoming the State Space Explosion Problem

It took the use of the dyamic reordering algorithm (via the '-dynamic' NuSMV option, see above) to overcome the State Space Explosion problem, and hence make pamtester-fm viable for use for production Linux-PAM configurations.

Table 27 shows the test results for testing of Linux-PAM configurations when using this algorithm. The improvement in NuSMV performance was dramatic.

[8] Note, the HCPN and NuSMV encoding has changed slightly since the test. The test shows results for the older HCPN and NuSMV encodings, where as the HCPN encoding shown in Figure 86 uses the improved encoding. In any case, the older and the newer encodings are similar in terms of net structure, and use the same net structure and NuSMV optimizations.

			Build Model					Model Check model	
# of "stacked" PAMS	# of state variables[10]	Read model, flatten hierarchy, Build flat model (seconds)	Time used (seconds)	RAM used (MB)	(# BDD nodes, in millions) (# clusters)	BDDs/Cluster (Cluster1) (Cluster2) . . . (Cluster6)	Transition Relation Totality Check (seconds)	Time used (seconds)	RAM used (MB)
10	N/R	N/R	1.3	N/R	N/R	N/R	0.2	N/R	N/R
20	N/R	N/R	2.8	N/R	N/R	N/R	2.6	N/R	N/R
40	N/R	N/R	9.2	12.2	N/R	N/R	27.5	1.2	N/R
80	N/R	0.1	34.1	N/R	N/R	N/R	88.3	8.3	N/R
160	1392	N/R	165.2	50.5	211182 19	N/R	N/R	N/R	N/R
320	2676	N/R	562.8	91.7	809828 35	N/R	N/R	N/R	N/R

Table 27: Overcoming State Space Explosion - Results Summary

Note that the amount of stacked PAMs is much larger than what is normally used for *production* Linux-PAM configurations. Here, our test Linux-PAM configuration reaches 320 "stacked" PAMs. In contrast, an average production Linux-PAM configuration (as found on Fedora Core 6 default Linux-PAM installation), was around 4 "stacked" PAMs!

Here, even at 320 "stacked" PAMs, we only use 91.7 MB of RAM and 562 seconds when building our transition system model (in contrast to already using 260 MB of RAM at 5 "stacked" PAMs, and using 664.8 seconds at 7 "stacked" PAMs, using the default NuSMV algorithm)!

Similarly, drastic improvements are seen, both in the transition relation totality check time, and in the model checking time.

[10] Note, the HCPN and NuSMV encoding has changed slightly since the test. The test shows results for the older HCPN and NuSMV encodings, where as the HCPN encoding shown in Figure 86 uses the improved encoding. In any case, the older and the newer encodings are similar in terms of net structure, and use the same net structure and NuSMV optimizations.

Further research, and a complete test suite needs to be carried out, and fully documented to make further comments.

None the less, these results are encouraging, especially when the test platform is a household PC – an off-the shelf Lenovo laptop with a Dual Core 2.6 Ghz CPU and 3 GB of RAM.

Results of Verification of Production Linux-PAM Configurations

Linux-PAM Client	Management Function	PAM Stack Instance size (in "stacked" Pams)	Transition System Model Verification	
			Generation of Transition System Model (in seconds)	Model Checking of Transition System Modeltime (in seconds)
sudo	pam_authenticate()	4	1.9	5.0
cups	pam_authenticate()	4	1.9	5.1
sshd	pam_authenticate()	4	2.0	5.1
smtp	pam_authenticate()	4	1.8	5.3
passwd	pam_authenticate()	4	2.0	5.3
login	pam_authenticate()	5	2.3	2.0
gdm	pam_authenticate()	5	2.4	2.3
su	pam_authenticate()	5	2.0	11.9
crond	pam_authenticate()	6	2.3	11.3
wireshark	pam_authenticate()	6	2.3	24.2
average		4.7	2.1	7.8

Table 28: Results of pamtester-fm verfication of 'security properties' of production Linux-PAM configurations

Based on the optimizations made in the HCPN and Transition System specifications, and after choosing a better-performing model checking algorithm, we used pamtester-fm to do specification and verification of several, production Linux-PAM configurations.

Specifically, pamtester-fm was used to specify and verify production Linux-PAM Configurations for the following Linux-PAM Clients: sudo, cups, sshd, smtp, passwd, login, gdm, su, crond and wireshark.

Only the results for Linux-PAM Configuration corresponding to the pam_authenticate() Management Function are refered to in this section.

Pamtester-fm was used to parse the corresponding Linux-PAM Configurations, generate the corresponding PAM Stack Instance HCPN models, and generate a NuSMV-syntax encoding of the corresponding Transition System.

Then, pamstester-fm executed the NuSMV program, instructing NuSMV to use dynamic reordering (via the '–dynamic' argument), with the encoded Transition Systems, and the "security properties" as input. The specifics of how pamtester-fm does this is outlined in section on page 139.

Table 28 outlines the results of the production Linux-PAM configuration testing.

DISCUSSION

It is useful to be able to obtain the set of possible PAM_RETURN values of a PAM Stack Execution, of an Effective PAM Stack Instance corresponding to an arbitrary Linux-PAM Configuration, for any Linux-PAM client program, for any Management Function.

Given the complexity of the operation and configuration of Linux-PAM, a modification of a single element of a Linux-PAM configuration can affect the structure and execution of the Effective PAM Stack instances of the Linux-PAM configuration. This in turn may affect the authentication-related functionality that is provided by the affected PAM Stacks. Sometimes, such a modification may also alter the set of possible PAM_RETURN values of an Effective PAM Stack instance.

For instance, suppose that before a modification, an Effective PAM Stack instance associated with pam_authenticate() for the login program is capable of returning PAM_SUCCESS = 0. This means that it is possible to successfully carry out user authentication (successfully authenticate to the system). Now, suppose that the administrator make a single modification to the Linux-PAM Configuration of the login program. Furthermore, suppose that this modification results in the Effective PAM Stack instance of pam_authenticate() to no longer be able to return PAM_SUCCESS = 0. Effectively, this modification resulted in a Denial of Service (DOS) condition, as now, users are not able to successfully authenticate to the system.

An automated tool like pamtester-fm, a tool that can automatically generate the set of all possible PAM_RETURN values of a PAM Stack instance, just by this functionality alone, already has a tremendous utility.

For example, in the above case, a Linux-PAM administrator can run pamtester-fm on the Linux-PAM Configuration before the configuration modification, and after the configuration modification. Comparing both results, the Linux-PAM administrator can see how a modification affects the set of possible PAM_RETURN values, including creation of a DOS condition. A DOS condition is a major form of an "attack" against an element of an information technology infrastructure. In this case, pamtester-fm can be used as a tool to detect DOS conditions, and hence help avoid this major type of "attack".

Supposing that the HCPN model, and the resulting transition system model are valid (i.e.accurately represent the PAM Stack execution process and the corresponding Linux-PAM Configuration), then such envisioned functionality can be achieved, as foreshadowed by the current state of the pamtester-fm tool. The usage of dynamic ordering of variables during the model checking phase created "breakthrough" results, where Linux-PAM Configurations containing 10s and 100s of PAMs were used, yet the model checking time was only seconds, ones, and tens of minutes. Also, the RAM requirements were negligible, as even the larger test Linux-PAM Configurations only used around 30 MB of RAM. Given the above, and the fact that average production Linux-PAM Configuration consisted of 6 modules, gives the modeler "room to breathe". Specifically, the model sophistication can be increased. For example, the HCPN models describing both the PAM Stack execution (i.e. see Flags, Frozen Chain, PAM Options, in

Limitations), and the execution of individual PAMs (i.e. Flags, PAM Options) can be made more complex, and hence better approximate their operation.

Another consequence of these results is that Linux-PAM Administrators may perhaps feel more "adventurous" in their modification and creation of Linux-PAM Configurations. Currently, major GNU/Linux distributions supply Linux-PAM Configurations for all of the major applications. Furthermore, it is not a common practice to make extensive modifications to these configurations, or create new configurations. Perhaps a tool like pamtester-fm will encourage Linux-PAM Administrators to start experimenting with modification and creation of Linux-PAM Configurations. For instance, none of the production Linux-PAM Configurations used in testing contained substacks, yet substack functionality is a feature of Linux-PAM PAM Stack execution. Perhaps this feature is not useful, or perhaps Linux-PAM is seen as too complex already, and introducing substacks into configurations is seen as an unnecessary hazard. Similarly, most Configuration Lines, within the production Linux-PAM configurations that were tested, were using Simple Controls. Yet, Linux-PAM offers the flexibility of Complex Controls. Simple Controls have equivalent Complex Controls. Thus, Simple Controls are just labels. These labels (i.e. optional, required, sufficient, requisite) are just somebody's semantic interpretation of their functionality. Why do Linux-PAM administrators not create their own semantic interpretations? Again, perhaps the current set of Basic Controls is sufficient. Or, perhaps, Linux-PAM is just too complex as it is, and modifying existing configurations or creating new configurations that use Complex Controls is seen as too dangerous.

Limitations

Pamtester-fm tool is a starting point, or a foundation for further modeling. As such, there are many limitations that still make pamtester-fm-generated models not close enough to being valid models of PAM Stack Executions.

PAM modeling is not detailed

Currently, the execution of PAM Module API function implementations, i.e. implementation of pam_sm_authenticate() of pam_unix.so, is modeled via a single HCPN transition. In the model, we define this transition to produce exactly one PAM_RETURN token, where the value of this token belongs to the set of possible return values of the Module API function implementation.

Thus, we only model a returning of a possible PAM_RETURN value, but we do not model the execution behaviour of a PAM. In other words, our modeling of an execution of a PAM solely consists of randomly returning a possible PAM return value. This modeling does not capture how the PAM functions, given external factors. For example, the pam_unix.so PAM may behave differently, and hence return only a subset of the possible PAM return values, if the user supplied password is blank. Currently, our model is not capable of differentiating this possibility.

PAM options are not modeled

In the Linux-PAM Configuration, the administrator can provide a list of PAM options. These options parameterize how the PAM functions.

Currently, our model does not implement the usage of options by PAMs.

Some Management Functions are not modeled

For instance, at this point, pamtester-fm does not create valid models for the PAM Stack Executions of PAM Stack Instances corresponding to the Management Functions:

- pam_setcred(),
- pam_chauthtok(), and
- pam_close_session().

This is because these functions use a feature of PAM Stack execution, called Freeze Chain. Freeze Chain functionality implements the "freezing" of a PAM Stack Instance during its execution. A "freezing" of a PAM Stack Instance occurs during the execution of the PAM Stack Intance. This "freezing" consists of storing the results of the corresponding "stacked" PAM's execution return values in the PAM Stack Instance. Once a PAM Stack is frozen, then, subsequent Management Function executions, called during the same Authentication Process, may use the "frozen" values (the PAM execution return values saved on the PAM Stack Instance), instead of the newly obtained PAM execution return values.

Implementing Frozen Chain in the model requires significant additions to the HCPN model. Due to this, modeling of Frozen Chain was omitted.

Flags are not modeled

All Management Functions accept a flags argument, i.e. int flags. A flags argument affects the way that these management functions are carried out, and the way that the

corresponding "stacked" PAMs provide authentication-related functionality. For example, for pam_authetnicate(), one flag that can be specified is called PAM_DISALLOW_NULL_AUTHTOK. When this flag is passed, this instructs Linux-PAM to return the PAM_AUTH_ERR = 7 error PAM Stack execution return value, if the user does not have an authentication token configured on the system, i.e. a blank password.

Flags affect how the executed PAMs behave. Specifically, these flags are passed onto the "stacked" PAMs during PAM Stack execution. The list of options is specified as an argument to the call made to the implementation of the Module API function of the corresponding Management Function of the "stacked" PAM. For example, supposing that PAM_DISALLOW_NULL_AUTHTOK is specified during the call to pam_authenticate() by the Linux-PAM client, then, when Linux-PAM executes the Effective PAM Stack instance for pam_authenticate(), Linux-PAM calls pam_sm_pam_authenticate() implementation of each "stacked" PAM and passes PAM_DISALLOW_NULL _AUTHTOK as one of the arguments.

Implementation of flag passing would require significant modeling. Also, since currently, the execution of PAM Module API function implementation is only modeled via a single HCPN transition, hence it would not be beneficial to model the passing of flags. This is because, mainly, flags parameterize the execution of Module API function implementations of PAMs. Thus, we chose to omit modeling of passing of flags, for now.

Future Work

The results of this thesis provide a foundation for a variety of future work. We discuss future work based on the following categories: improving modeling approximation to Linux-PAM; identification of prevalent Linux-PAM configuration scenarios; formalizing a logic, algorithms, and techniques for Linux-PAM configuration analysis; attacks and defences from an Information Security perspective; interoperability with other formal methods software; end-user pamester-fm tool interaction; and next-generation interactive Linux-PAM systems.

Improving modeling approximation to Linux-PAM

Pamtester-fm lays the foundation for further modeling. PAM modeling needs to become more sophisticated. PAM Stack Execution also needs to be modeled further. Passing of flags, PAM Options and Frozen Chain are just some elements that need to be addressed with further modeling.

Identification of Common Linux-PAM Configuration Scenarios

A set of commonly used PAM Configurations should be identified. This set should be obtained from the usage patterns of the Linux-PAM community. For example, there should be a common, prevalent configuration for Linux-PAM authentication with bindings to a Windows LDAP/Active Directory server, for example. Once identified, these configurations should be modeled thoroughly, further increasing the utility of pamtester-fm as a tool in the Linux-PAM administrator's tool box.

Developing Algebraic Techniques for Linux-PAM Configuration Analysis

A Linux-PAM installation, the Linux-PAM Configurations, the operation of Linux-PAM clients, the operation of Linux-PAM PAMs, and the surrounding subsystems (i.e. GNU C Library configuration of user accounts), form an intricate dependency system. This dependency system can be viewed from different perspectives: as a combination of algebraic structures, or a composition of subsystems and the protocols that bind them together.

For example, an Effective PAM Stack Instance is a sequence of PAMs. Hence an Effective PAM Stack Instance is a sequence. Given two sequences x and y, and a function, say *, that operates on sequences, what do we get by combining x with y using this operation? Precisely, what is x * y? What are the properties of x * y? Does x = y? What is =? Does x * y = y * x? Given x, y, z, does (x * y) * z = x * (y * z)? Is there a Linux-PAM configuration that can be viewed as a unity, i.e. x * 1 = x? Questions abound. In another example, an Effective PAM Stack Instance is a sequence of PAMs. Each PAM in this sequence is, in a way, dependent on the PAMs preceding it. Suppose + is a function that stack PAMs together to form PAM Stack Instances. Then, in what ways can we characterize PAM "stacking"? For example, for which PAMs is it the case that a + a = a? Would such a system be useful? Could we leverage this notation and the mechanics behind to study Linux-PAM systems? I.e. x * y = (a + b + c) * (d + e) = … etc.

From the view of subsystems and protocols, individual PAMs may embody functionality that is protocol-like in a way that the PAM operates, or interacts with other

PAMs or surrounding components. Certainly, some PAMs use protocols to interact with servers, i.e. LDAP, Kerberos or UNIX authentication PAMs.

Also, Linux-PAM and Linux-PAM Clients have a standardized protocol defined for Linux-PAM to Linux-PAM Client communication! This protocol ensures that no matter how the Linux-PAM Client is written, the Linux-PAM Client can provide the necessary information back to Linux-PAM. For example, whether one is using the login program (a text-based program) or an X-Window system based GUI program requiring authentication, the authentication portion for both of these programs has to provide a username and a password back to Linux-PAM. In order to do this, Linux-PAM uses the notion of a Linux-PAM Conversation. This Conversation has a protocol-like standardized definition.

Attacks and Defences from an Information Security Perspective

Just because Linux-PAM is not inherently about cryptography or a communication protocol, does not mean that it does not fall under the umbrella of Information Security. On the contrary, the problem of specifying and verifying Linux-PAM configurations is at the heart of information security. This is because Linux-PAM configurations protect system access.

Due to this, Linux-PAM configurations, and its surrounding environment (i.e. OS parameters such as existence of a user account), can be studied as an attack surface. In this case, an unwitting Linux-PAM administrator is also considered to be an "attacker".

One such attack is outlined below.

Unauthorized User Access

We introduce a class of attacks on a Linux-PAM installation, called the *Linux-PAM Configuration Modification* (LCM) attack. An attack of type LCM occurs when a Linux-PAM administrator makes a modification to the existing Linux-PAM configuration, causing a change to the set of possible PAM Stack executions, where one of these PAM Stack executions does not satisfy some set of "security properties".

One instance of an LCM attack causes unauthorized user access to the system.

For example, given a Linux-PAM Configuration for the login program, suppose that it is possible for users to successfully authenticate to the system (a necessary step before a user can log on to a system, as implemented by login). Now, suppose that the Linux-PAM administrator makes a modification to the Linux-PAM Configuration of the login program. Further, suppose that this modification still allows successful authentication, but, the user in question can now be an unauthorized user – a user without a username or password configured on the system in question. This is certainly possible. To give a trivial example, a Linux-PAM Configuration providing successful authentication for an unauthorized user is shown in Figure 88 .

```
login auth required pam_allow.so
```

Figure 88: Linux-PAM Configuration enabling unauthorized user access

Here, the Effective PAM Stack Instance for pam_authenticate() contains exactly one "stacked" PAM – pam_allows.so PAM. The functionality of pam_allow.so PAM consists of simply returning PAM_SUCCESS, for all Management Functions. Furthermore, for

the pam_authenticate() function, if the principal does not supply a username when attempting to authenticate (i.e. an empty username string is provided to the login program), the pam_allow.so PAM assigns a DEFAULT_USER = "nobody" username, and then returns PAM_SUCCESS = 0.

Although this example is trivial, the point is to demonstrate that a principal (an external-to-the-system entitity) that does not have a valid user account (and password) on the system can still successfully authenticate. Actually, such a situation is not far fetched, as due to the complexity of Linux-PAM Configuration parsing, it is not difficult to make a mistake (file name creation errors, file location errors, include or substack specification errors, syntax errors, reaching maximum substack level – root causes which may result in this condition) where the Linux-PAM configuration parsing results in such a "trivial" Linux-PAM Configuration[11].

Thus, future work can involve creating models where we can detect the conditions that result in unauthorized user access. In this case, we have to model the notion of an "unauthorized user" as Linux-PAM interprets it. This can be done by in-depth modeling of PAMs that do user authentication. For example, pam_unix.so PAM does user authentication. Pam_unix.so PAM makes certain GNU C Library function calls which determine whether or not the principal has a user account on the system. We can abstract such calls and model them with HCPN constructs. For example, say we create an HCPN place called 'pPAM_UNIX-authorized_user' and define its colour set to be a CONTROL

[11] Recall, Linux-PAM continues to operate even if an error is encountered. Erronous Linux-PAM configurations continue to be used in provision of authentication-related functionality to Linux-PAM clients.

token. Then, semantically, we can define the model such that a CONTROL token is present in the HCPN place pPAM_UNIX-authorized_user, if, and only if, the principal has a user account on the system (and hence is an authorized user by our definition). This way, we can introduce the notion of an "authorized user" into our model.

Consequently, when we do model checking, we can employ "security property" specifications which use the value of pPAM_UNIX-authorized_user to determine whether or not the principal is an authorized user. Based on this, we can create formal specifications of "security properties" which involve the notion of an "authorized user".

Using this approach, we can create automated specifications and verifications of Linux-PAM configurations, where we can check if, given a Linux-PAM configuration, there exists a possible PAM Stack Execution where an "unauthorized user" can successfully authenticate, i.e. "AG (p_End != 0 | pPAM_UNIX-authorized_user == 1)".

Ultimately, given this capability, the pamtester-fm tool can be used as an Information Security auditing tool. For example, during penetration testing, the pamtester-fm tool can be utilized to check if there is a possibility that a Linux-PAM installation allows unauthorized user access for some application.

Interoperability with Other Formal Methods Software

Ability for pamtester-fm to utilize different model checking software packages should be incorporated. For example, plug-ins could be written, where each plug-in is dedicated to a single model checking software package.

Ideally, pamtester-fm should not have anything to do with doing the actual model instantiation and model checking. The focus of pamtester-fm should be on Linux-PAM configurations, i.e. parsing of Linux-PAM configurations, and based on this parsing, generating the appropriate model specification. Once that is done, the model generation and checking should be left to software tools that are made specifically for that task. These software tools must have functionality that allows pamtester-fm to control them programmatically, and for pamtester-fm to be able to obtain output of their analysis.

Ideally, an HCPN export function should be developed. Based on a Linux-PAM Configuration, Pamtester-fm would generate the HCPN module specification, and then this export function would encode it into an HCPN format that can be understood by another, external HCPN software package, which can then run analysis on this HCPN, including reachability analysis.

Currently pamtester-fm specifies and generates the HCPN model, based on its parsing of a Linux-PAM Configuration.

Currently pamtester-fm encodes the HCPN model instance into a transition system specification.

Currently pamtester-fm specifies the transition system model.

Currently, pamtester-fm uses NuSMV to generate and model check the transition system model. In this case, pamtester-fm provides NuSMV with the transition system model specification.

End-User Pamester-fm Tool Interaction

The success of pamtester-fm tool depends on the end-user experience. The difficulty here is how to create a "push-button" technology that does not require the end-user to understand the underlying formal methods concepts, yet provides meaningful results. In particular, how do we present the complexity and operation of PAM Stack Instance execution and its troubleshooting (model-checking) in an intuitive way that provides meaningful results back to the end-user? This may be achieved with a GUI.

Besides presentation of operation and results of PAM Stack Executions, there are other aspects that are dependent on the interface between the end-user and pamtester-fm. For example, the end-user should be able to parameterize the models of PAM Stack Execution, if needed. For instance, the end-user should be able to specify that the PAM Stack Execution should assume that the modeled authentication-related task (i.e. user authentication), had the user provide a blank password.

Lastly, the pamtester-fm interface should leverage the power of today's visualization and cheap display technologies. Specifically, PAM Stack Execution visualizations showing the "stacked" PAMs, and possible PAM Stack executions, real-time, or post-mortem could be done. The applications for this could be live system monitoring as well as information security visualization.

Next-Generation Interactive Linux-PAM Systems

Lastly, using all of above future work items as a foundation, next-generation interactive Linux-PAM systems could be developed. For example, such systems could do

real-time, reactive re-configuration of production Linux-PAM configurations, combined with visualization and monitoring capabilities.

CONCLUSION

One can use formal specification and verification techniques to solve a variety of problems. For example, in (6), the (sophisticated) ferryman uses the method of model checking a Transition System to ensure that his goods (a wolf, a goat, and a cabbage) are all safely transported to the other side of the river. In our case, pamtester-fm formally specifies and verifies that all possible PAM Stack execution sequences satisfy a set of "security properties" of the form: "it is not possible for a PAM Stack Instance to return PAM_RETURN = x", where x is an integer between 0 and 31.

Formal specification and verification approaches do not have to be, and neither should one expect them to be, "silver bullets" (tools that solve a problem completely, and with no error). For example, before an Information Technology industry presentation about the pamtester-fm.org project, the presentation organizer playfully asked: "so, is pamtester-fm going to make my Linux 100% secure?" This comment sheds light on the existence of an IT industry view that formal methods are expected to completely, and with no error, solve a problem they undertake. If this was not bad enough, additionally, the general feeling amongst IT industry practitioners is that formal methods are: difficult to understand, and too impractical (in terms of money and time investment) to apply to real-world problems. These views were the driving factors behind this thesis as a whole, and the development of the pamtester-fm tool and the pamtester-fm.org project – proof-of-concept work done as part of this thesis work.

The solutions generated by such tools do not have to fully, and with no error, solve a problem. As long as a portion of a problem is addressed, and this is useful, then progress

has been made. In this case, pamtester-fm "uncovers" the complexity of PAM Stack executions – a complexity which many Linux-PAM administrators are either unaware of, or are unable to, or do not bother to, cope with.

The creation of such tools does not have to be constrained by the limitations of existing formal specification and verification software. As long as the theory is correctly implemented, editor software, a compiler, and a suite of helper applications can suffice. In our work, pamtester-fm was written using the C programming language, and utilized other open source projects, such as NuSMV, GraphViz and GNU/Linux. The existing formal methods software tools were found to contain limitations which disqualified them from being used for this thesis. For example, a suite of existing tools could not be found that would automatically do the job of: accepting an arbitrary set of HCPN module specifications, combine this set to form a single HCPN, model check the behaviour of this resulting HCPN, and produce textual output. This forced the development of a custom tool that would do this job. Albeit, fortunately and critically, the NuSMV software package was located, which accepts arbitrary transition system specifications as input, automatically builds and model checks this transition system, and outputs the results of this check in textual format.

Pamtester-fm provides a foundation on which further modeling can be pursued. Based on the initial results, it is the opinion of the author of this thesis that software projects, including ones in the area of Information Security, can benefit from formal specification and verification methods. Again, they do not have to "promise" a system that is 100% "secure", yet their functionality can still be of benefit. In particular, pamtester-fm

demonstrates how a complex software system can be abstracted to the point where modeling of this system is manageable, yet useful. One interesting observation of this work was that when we model behaviour of a software system, we don't need to model every variable, or the whole function call chain. In many cases, it is sufficient to model the first one or two function call levels, and then, only to model a subset of variables. Often, it is enough to use a single HCPN transition or a single HCPN place to model a whole function call chain. Also, ignoring or abstracting away portions of the software system does not take away from the validity of the model. Based on this work, perhaps it would be useful for software projects to make use of personnel for the purpose of formal specification and verification.

Lastly, it is the opinion of the author of this thesis that opportunities exist to employ formal methods. Specifically, automated tools, that are based on formal methods, yet hide the complexity of formal methods to the tool's end-users, can assist IT industry professionals in carrying out regular job duties. In particular, tasks requiring solving repetitive, well-defined problems, too tedious for humans, yet "easy" enough to create formal models for, lend well to being approached with such tools. For example, enumerating all possible PAM Stack Executions, for all Effective PAM Stacks, of an arbitrary Linux-PAM Configuration, for a list of 10s of Linux-PAM Clients, and doing this every 5 minutes in a 24/7/365 environment, is an ideal candidate for such a formal methods approach. In fact, such approaches may act as a catalyst for next-generation Information Security tools - tools for which demand is likely to increase as our society becomes increasingly dependent on information technology infrastructures.

APPENDIX A: Source Code of PAM Stack Instance Execution

Pam_dispatch() from libpam/pam_dispatch.c

Pam_dispatch() chooses the Effective PAM Stack Instance, and then calls pam_dispatch_aux() to initiate the execution of this Effective PAM Stack Instance.

```
/*
 * This function translates the module dispatch request into a pointer
 * to the stack of modules that will actually be run.  the
 * _pam_dispatch_aux() function (above) is responsible for walking the
 * module stack.
 */

int _pam_dispatch(pam_handle_t *pamh, int flags, int choice)
{
    struct handler *h = NULL;
    int retval, use_cached_chain;
    _pam_boolean resumed;

    IF_NO_PAMH("_pam_dispatch", pamh, PAM_SYSTEM_ERR);

    if (__PAM_FROM_MODULE(pamh)) {
   D(("called from a module!?"));
   return PAM_SYSTEM_ERR;
    }

    /* Load all modules, resolve all symbols */

    if ((retval = _pam_init_handlers(pamh)) != PAM_SUCCESS) {
   pam_syslog(pamh, LOG_ERR, "unable to dispatch function");
   return retval;
    }

    use_cached_chain = _PAM_PLEASE_FREEZE;

    switch (choice) {
    case PAM_AUTHENTICATE:
   h = pamh->handlers.conf.authenticate;
   break;
    case PAM_SETCRED:
   h = pamh->handlers.conf.setcred;
   use_cached_chain = _PAM_MAY_BE_FROZEN;
   break;
    case PAM_ACCOUNT:
   h = pamh->handlers.conf.acct_mgmt;
   break;
    case PAM_OPEN_SESSION:
   h = pamh->handlers.conf.open_session;
   break;
    case PAM_CLOSE_SESSION:
   h = pamh->handlers.conf.close_session;
   use_cached_chain = _PAM_MAY_BE_FROZEN;
   break;
    case PAM_CHAUTHTOK:
   h = pamh->handlers.conf.chauthtok;
   if (flags & PAM_UPDATE_AUTHTOK) {
       use_cached_chain = _PAM_MUST_BE_FROZEN;
   }
   break;
```

```
    default:
   pam_syslog(pamh, LOG_ERR, "undefined fn choice; %d", choice);
   return PAM_ABORT;
    }

    if (h == NULL) {     /* there was no handlers.conf... entry; will use
          * handlers.other... */
   switch (choice) {
   case PAM_AUTHENTICATE:
       h = pamh->handlers.other.authenticate;
       break;
   case PAM_SETCRED:
       h = pamh->handlers.other.setcred;
       break;
   case PAM_ACCOUNT:
       h = pamh->handlers.other.acct_mgmt;
       break;
   case PAM_OPEN_SESSION:
       h = pamh->handlers.other.open_session;
       break;
   case PAM_CLOSE_SESSION:
       h = pamh->handlers.other.close_session;
       break;
   case PAM_CHAUTHTOK:
       h = pamh->handlers.other.chauthtok;
       break;
   }
    }

    /* Did a module return an "incomplete state" last time? */
    if (pamh->former.choice != PAM_NOT_STACKED) {
   if (pamh->former.choice != choice) {
       pam_syslog(pamh, LOG_ERR,
             "application failed to re-exec stack [%d:%d]",
             pamh->former.choice, choice);
       return PAM_ABORT;
   }
   resumed = PAM_TRUE;
    } else {
   resumed = PAM_FALSE;
    }

    __PAM_TO_MODULE(pamh);

    /* call the list of module functions */
    pamh->choice = choice;
    retval = _pam_dispatch_aux(pamh, flags, h, resumed, use_cached_chain);
    resumed = PAM_FALSE;

    __PAM_TO_APP(pamh);

    /* Should we recall where to resume next time? */
    if (retval == PAM_INCOMPLETE) {
   D(("module [%d] returned PAM_INCOMPLETE"));
   pamh->former.choice = choice;
    } else {
   pamh->former.choice = PAM_NOT_STACKED;
    }

    return retval;
}
```

Figure 89: Source Code of pam_dispatch() from libpam/pam_dispatch.c

Pam_dispatch_aux() from libpam/pam_dispatch.c

Pam_dispatch_aux() does the execution of the Effective PAM Stack Instance, as described by the PAM Stack Execution algorithm in Table 13 on page 29.

```
/*
 * walk a stack of modules.  Interpret the administrator's instructions
 * when combining the return code of each module.
 */

static int _pam_dispatch_aux(pam_handle_t *pamh, int flags, struct handler *h,
              _pam_boolean resumed, int use_cached_chain)
{
    int depth, impression, status, skip_depth, prev_level, stack_level;
    struct _pam_substack_state *substates = NULL;

    IF_NO_PAMH("_pam_dispatch_aux", pamh, PAM_SYSTEM_ERR);

    if (h == NULL) {
        const void *service=NULL;

   (void) pam_get_item(pamh, PAM_SERVICE, &service);
   pam_syslog(pamh, LOG_ERR, "no modules loaded for `%s' service",
         service ? (const char *)service:"<unknown>" );
   service = NULL;
   return PAM_MUST_FAIL_CODE;
    }

    /* if we are recalling this module stack because a former call did
       not complete, we restore the state of play from pamh. */
    if (resumed) {
   skip_depth = pamh->former.depth;
   status = pamh->former.status;
   impression = pamh->former.impression;
   substates = pamh->former.substates;
   /* forget all that */
   pamh->former.impression = _PAM_UNDEF;
   pamh->former.status = PAM_MUST_FAIL_CODE;
   pamh->former.depth = 0;
   pamh->former.substates = NULL;
    } else {
   skip_depth = 0;
   substates = malloc(PAM_SUBSTACK_MAX_LEVEL * sizeof(*substates));
   if (substates == NULL) {
       pam_syslog(pamh, LOG_CRIT,
             "_pam_dispatch_aux: no memory for substack states");
       return PAM_BUF_ERR;
   }
   substates[0].impression = impression = _PAM_UNDEF;
   substates[0].status = status = PAM_MUST_FAIL_CODE;
    }

    prev_level = 0;

    /* Loop through module logic stack */
    for (depth=0 ; h != NULL ; prev_level = stack_level, h = h->next, ++depth) {
   int retval, cached_retval, action;

        stack_level = h->stack_level;

   /* skip leading modules if they have already returned */
   if (depth < skip_depth) {
       continue;
```

```
    }

    /* remember state if we are entering a substack */
    if (prev_level < stack_level) {
        substates[stack_level].impression = impression;
        substates[stack_level].status = status;
    }

    /* attempt to call the module */
    if (h->handler_type == PAM_HT_MUST_FAIL) {
        D(("module poorly listed in PAM config; forcing failure"));
        retval = PAM_MUST_FAIL_CODE;
    } else if (h->handler_type == PAM_HT_SUBSTACK) {
        D(("skipping substack handler"));
        continue;
    } else if (h->func == NULL) {
        D(("module function is not defined, indicating failure"));
        retval = PAM_MODULE_UNKNOWN;
    } else {
        D(("passing control to module..."));
        pamh->mod_name=h->mod_name;
        retval = h->func(pamh, flags, h->argc, h->argv);
        pamh->mod_name=NULL;
        D(("module returned: %s", pam_strerror(pamh, retval)));
    }

    /*
     * PAM_INCOMPLETE return is special.  It indicates that the
     * module wants to wait for the application before continuing.
     * In order to return this, the module will have saved its
     * state so it can resume from an equivalent position when it
     * is called next time.  (This was added as of 0.65)
     */
    if (retval == PAM_INCOMPLETE) {
        pamh->former.impression = impression;
        pamh->former.status = status;
        pamh->former.depth = depth;
        pamh->former.substates = substates;

        D(("module %d returned PAM_INCOMPLETE", depth));
        return retval;
    }

    /*
     * use_cached_chain is how we ensure that the setcred/close_session
     * and chauthtok(2) modules are called in the same order as they did
     * when they were invoked as auth/open_session/chauthtok(1). This
     * feature was added in 0.75 to make the behavior of pam_setcred
     * sane. It was debugged by release 0.76.
     */
    if (use_cached_chain != _PAM_PLEASE_FREEZE) {

        /* a former stack execution should have frozen the chain */

        cached_retval = *(h->cached_retval_p);
        if (cached_retval == _PAM_INVALID_RETVAL) {

       /* This may be a problem condition. It implies that
          the application is running setcred, close_session,
          chauthtok(2nd) without having first run
          authenticate, open_session, chauthtok(1st)
          [respectively]. */

       D(("use_cached_chain is set to [%d],"
          " but cached_retval == _PAM_INVALID_RETVAL",
          use_cached_chain));
```

```
      /* In the case of close_session and setcred there is a
         backward compatibility reason for allowing this, in
         the chauthtok case we have encountered a bug in
         libpam! */

      if (use_cached_chain == _PAM_MAY_BE_FROZEN) {
          /* (not ideal) force non-frozen stack control. */
          cached_retval = retval;
      } else {
          D(("BUG in libpam -"
             " chain is required to be frozen but isn't"));

          /* cached_retval is already _PAM_INVALID_RETVAL */
      }
       }
   } else {
       /* this stack execution is defining the frozen chain */
       cached_retval = h->cached_retval = retval;
   }

   /* verify that the return value is a valid one */
   if ((cached_retval < PAM_SUCCESS)
       || (cached_retval >= _PAM_RETURN_VALUES)) {

       retval = PAM_MUST_FAIL_CODE;
       action = _PAM_ACTION_BAD;
   } else {
       /* We treat the current retval with some respect. It may
          (for example, in the case of setcred) have a value that
          needs to be propagated to the user.  We want to use the
          cached_retval to determine the modules to be executed
          in the stacked chain, but we want to treat each
          non-ignored module in the cached chain as now being
          'required'. We only need to treat the,
          _PAM_ACTION_IGNORE, _PAM_ACTION_IS_JUMP and
          _PAM_ACTION_RESET actions specially. */

       action = h->actions[cached_retval];
   }

   D(("use_cached_chain=%d action=%d cached_retval=%d retval=%d",
      use_cached_chain, action, cached_retval, retval));

   /* decide what to do */
   switch (action) {
   case _PAM_ACTION_RESET:

       impression = substates[stack_level].impression;
       status = substates[stack_level].status;
       break;

   case _PAM_ACTION_OK:
   case _PAM_ACTION_DONE:

       if ( impression == _PAM_UNDEF
       || (impression == _PAM_POSITIVE && status == PAM_SUCCESS) ) {
                /* in case of using cached chain
                   we could get here with PAM_IGNORE - don't return it */
                if ( retval != PAM_IGNORE || cached_retval == retval ) {
          impression = _PAM_POSITIVE;
                    status = retval;
                }
       }
       if ( impression == _PAM_POSITIVE && action == _PAM_ACTION_DONE ) {
      goto decision_made;
       }
       break;
```

```
   case _PAM_ACTION_BAD:
   case _PAM_ACTION_DIE:
#ifdef PAM_FAIL_NOW_ON
       if ( cached_retval == PAM_ABORT ) {
      impression = _PAM_NEGATIVE;
      status = PAM_PERM_DENIED;
      goto decision_made;
       }
#endif /* PAM_FAIL_NOW_ON */
       if ( impression != _PAM_NEGATIVE ) {
      impression = _PAM_NEGATIVE;
           /* Don't return with PAM_IGNORE as status */
           if ( retval == PAM_IGNORE )
          status = PAM_MUST_FAIL_CODE;
      else
          status = retval;
       }
       if ( action == _PAM_ACTION_DIE ) {
      goto decision_made;
       }
       break;

   case _PAM_ACTION_IGNORE:
       break;

        /* if we get here, we expect action is a positive number --
           this is what the ...JUMP macro checks. */

   default:
       if ( _PAM_ACTION_IS_JUMP(action) ) {

      /* If we are evaluating a cached chain, we treat this
         module as required (aka _PAM_ACTION_OK) as well as
         executing the jump. */

      if (use_cached_chain) {
          if (impression == _PAM_UNDEF
         || (impression == _PAM_POSITIVE
             && status == PAM_SUCCESS) ) {
                if ( retval != PAM_IGNORE || cached_retval == retval ) {
             impression = _PAM_POSITIVE;
                        status = retval;
                }
          }
      }

      /* this means that we need to skip #action stacked modules */
      while (h->next != NULL && h->next->stack_level >= stack_level && action > 0) {
          do {
         h = h->next;
         ++depth;
          } while (h->next != NULL && h->next->stack_level > stack_level);
          --action;
      }

      /* note if we try to skip too many modules action is
                  still non-zero and we snag the next if. */
       }

       /* this case is a syntax error: we can't succeed */
       if (action) {
      pam_syslog(pamh, LOG_ERR, "bad jump in stack");
      impression = _PAM_NEGATIVE;
      status = PAM_MUST_FAIL_CODE;
       }
   }
```

```
    continue;

decision_made:      /* by getting  here we have made a decision */
    while (h->next != NULL && h->next->stack_level >= stack_level) {
        h = h->next;
        ++depth;
    }
     }

     /* Sanity check */
     if ( status == PAM_SUCCESS && impression != _PAM_POSITIVE ) {
    D(("caught on sanity check -- this is probably a config error!"));
    status = PAM_MUST_FAIL_CODE;
     }

     free(substates);
     /* We have made a decision about the modules executed */
     return status;
}
```

Figure 90: Source Code of pam_dispatch_aux() from libpam/pam_dispatch.c

APPENDIX B: Authentication using 'login' at ACME Corp

Authentication Policy

```
ACME Corporation: User Authentication Security Policy

------------------------------------------------------------------------------

This is a policy for user authentication for the 'login' application.

The 'login' application allows a user to log on to a system using a computer terminal.

------------------------------------------------------------------------------

Step 1: If the user is root, and the user is not using a "secure" terminal, then deny system log on,
and exit the authentication process.

Step 2: Optionally set environment variables.

Step 3: Authenticate the user. If the user does not authenticate successfully, then deny system log
on.
```

Figure 91: A Linux-PAM Authentication Policy P

Linux-PAM Configuration

The Authentication Policy P shown in Figure 91 is interpreted and translated into a Linux-PAM Configuration. This Linux-PAM Configuration is specified in Linux-PAM Configuration syntax, and it is stored in the file /etc/pam.d/login (Figure 92). For the Service 'login', Linux-PAM interprets the file /etc/pam.d/login as the Root of the Client-Specific Linux-PAM Configuration for the Service 'login'. Hence, when Linux-PAM creates the corresponding PAM Stack Instances, /etc/pam.d/login is parsed as the first (and in this case, the only) file.

```
auth requisite pam_securetty.so
auth optional pam_env.so
auth sufficient pam_unix.so
auth required pam_deny.so
```

Figure 92: A Client-Specific Linux-PAM Configuration C, /etc/pam.d/login, for the Service 'login', implementing the Authentication Policy P

Figure 93 shows the implicit and explicit configuration line tokens after Linux-PAM parses the Client-Specific Linux-PAM Configuration contained in Figure 92. Implicitly, the Service token is interpreted by Linux-PAM to be 'login'. Also, all Simple Controls are shown as their equivalent Complex Controls.

```
login auth [success=ok, new_auth_tok_reqd=ok ignore=ignore default=die] pam_securetty.so
login auth [success=ok, new_auth_tok_reqd=ok default=ignore] pam_env.so
login auth [success=done, new_auth_tok_reqd=done, default=ignore] pam_unix.so
login auth [success=ok, new_auth_tok_reqd=ok, ignore=ignore, default=bad] pam_deny.so
```

Figure 93: The parsed Client-Specific Linux-PAM Configuration C for the Service 'login', implementing the Authentication Policy P

PAM Stack Specification

depth; i	*level*; L_i	*SERVICE*; S_i	*SERVICE GROUP*; G_i	*CONTROL*; C_i		*PATH*; P_i	*OPTIONS*; O_i
				$X \subseteq R(C_i)$	$d \in D(C_i)$: $\forall x \in X$: $C_i(x) = d$		
0	0	login	auth	0,12	-1	pam_securetty.so	
				25	0		
				"ow"	-4		
1	0	login	auth	0,12	-1	pam_env.so	
				"ow"	0		
2	0	login	auth	0,12	-2	pam_unix.so	
				"ow"	0		
3	0	login	auth	0,12	-1	pam_deny.so	
				25	0		
				"ow"	-3		

Table 29: Generation of PAM Stack Specification for the Authentication Management group for Service "login" with Complex Control equivalents

PAM Stack Instance

depth; i		*level*; L_i	*SERVICE*; S_i	*SERVICE GROUP*; G_i	*CONTROL*; C_i			*PATH*; P_i	*OPTIONS*; O_i
					$X \subseteq R(C_i)$	$d \in D(C_i)$: $\forall x \in X$: $C_i(x) = d$			
	int handler_type	int stack_level			int actions[32]		int (*func)	char *mod_name	char **argv
0	0	0	login	auth	0,12	-1	I(pam_sm_authenticate(), pam_securetty.so)	pam_securetty.so	
					25	0			
					"ow"	-4			
1	0	0	login	auth	0,12	-1	I(pam_sm_authenticate(), pam_env.so)	pam_env.so	
					"ow"	0			
2	0	0	login	auth	0,12	-2	I(pam_sm_authenticate(), pam_unix.so)	pam_unix.so	
					"ow"	0			
3	0	0	login	auth	0,12	-1	I(pam_sm_authenticate(), pam_deny.so)	pam_deny.so	
					25	0			
					"ow"	-3			

Figure 94: Generation of a PAM Stack Instance for the pam_sm_authenticate() Module API function for the Service "login"

Bibliography

1. **Morgan, A.** Linux-PAM Homepage. [Online] http://www.kernel.org/pub/linux/libs/pam/.

2. *Unified Login With Pluggable Authentication Modules (PAM), Request For Comments 86.0.* **Samar, V and Schemers, R.** s.l. : Open Software Foundation, 1995.

3. **Samar, V and Lai, C.** *Making Login Services Independent of Authentication Technologies.* 1996.

4. **Cavada, R.** NuSMV 2.4 User Manual. *NuSMV Model Checker.* [Online] 2005. http://nusmv.irst.itc.it.

5. **Jensen, K and Kristensen, L M.** *Coloured Petri Nets.* Berlin : Springer-Verlag, 2009.

6. **Huth, M and M, Ryan.** *Logic in Computer Science.* Cambridge : Cambridge University Press, 20066.

7. The Linux-PAM Module Writer's Guide. [Online] 2008. http://www.kernel.org/pub/linux/libs/pam/.