Techniques for Automating Policy Specification for Application-oriented Access Controls

Blinded.

Abstract—**By managing the authority assigned to each application, rule-based application-oriented access controls can significantly mitigate the threats posed by malicious code due to software vulnerabilities or malware. However, these policies are typically complex and difficult to develop. Learning modes can ease specification; however, they still require high levels of expertise to utilise correctly, and are most suited to confining non-malicious software.**

This paper presents a novel approach to automating policy specification for rule-based application-oriented access controls. The functionality-based application confinement (FBAC) model provides reusable parameterised abstractions. A number of straightforward yet effective techniques are presented that use these functionality-based abstractions to create application policies *a priori*; that is, without running programs before policies are specified. These techniques automate the specification of policy details by analysing program dependencies, program management information, and filesystem contents.

Keywords-application-oriented access control; policy automation; a priori policy specification; policy abstraction; functionality-based application confinement; sandboxing

I. INTRODUCTION

Application-oriented access controls have the potential to overcome shortcomings of traditional user-oriented approaches to access control. By managing the authority granted to applications, rather than to users, application-oriented access controls can limit the damage from applications acting maliciously due to malware or software vulnerabilities.

Isolation-based application-oriented access controls — such as traditional sandboxes [1, 2], virtual machines [3, 4], and containers [5, 6] — isolate all of the confined programs within a restricted environment to only access a simple pre-defined set of resources accessible from within that sandbox. Although isolation-based approaches are relatively easy to initially configure, they do not typically allow confined applications with different privilege requirements to interact without circumventing the isolation mechanism, and do not suit typical user workflows: for example, using one program to create a file, another to edit it, and a third program to share the file.

In contrast, rule-based application-oriented access controls define what each application is authorised to do, typically in terms of finely-grained privileges. This approach allows applications to cooperate and share resources, while confining each application to only perform the actions and access the resources that are required for the program to function as expected. Examples of rule-based schemes include Janus [7], Systrace [8], AppArmor [9], SELinux [10], and TOMOYO [11]. Rule-based application restrictions can greatly mitigate the threat posed by malicious code, and these approaches can avoid many of the limitations of isolation-based approaches. However, finely grained controls typically result in policy management complexity, leading to usability problems. Consequently these systems often rely on learning modes to develop policy; although, as discussed in the next section, learning modes themselves can pose problems.

This paper presents a number of automation techniques, unique within the field, that can ease the specification of rule-based application-oriented access control policies. This approach to automation leverages advantages of a functionality-based approach to policy [12] (such as reusable parameterised policy abstractions [13]) to provide *a priori* policy specification. That is, the construction of policies without having to first run the application to be confined. These techniques can improve the usability of rule-based application-oriented access controls, and enable end users to specify policies to protect themselves against the programs they run [14].

A Linux-based implementation, known as FBAC-LSM, has been developed, which demonstrates the efficacy of these techniques [15].

II. APPLICATION CONFINEMENT AND AUTOMATION

Previous application-oriented access controls, including Systrace [8], AppArmor [9], SELinux [10], and TOMOYO [11], typically rely on learning mode tools in order to create policies to confine applications. Learning mode tools are the most common approach to mitigating the usability problems associated with the policy complexity of finely-grained application-oriented access controls. These tools generate policy rules based on observing the actions of the programs that are to be confined. Generating policy in this way can usually be performed either while running the program unconfined, or while a policy is enforced. Either way, there are a number of drawbacks to this approach of constructing policy. Typically, rules are created while the program is unconfined. However, in this case, if the program behaves maliciously, then the program can do damage to the system and expose users' files to misuse while the policy is being constructed. On the other hand, if the program is being confined while policy is created, the program will typically need to be run many times in order to identify all the rules required in order for the program to successfully load and execute. In either case, the user creating the policy must carefully vet each of these rules to ensure that the program does indeed require access to the resources it tries to access. A usability study has shown that many end-users do not have the expertise required to correctly utilise learning modes [14].

The functionality-based application confinement (FBAC) model has been proposed as a way of modelling the authority granted to applications using reusable policy abstractions [12]. Policy abstractions, known as *functionalities*, are used to assign authority to applications based on the features they provide. Functionalities authorise elaborate sets of finely grained privileges based on high-level security goals, and adapt to the needs of specific applications through parameterisation.

The FBAC model separates the bulk of program analysis and rule specification from the task of specifying policies for individual applications. Since FBAC application policies are constructed using high-level abstractions, use of a learning mode is not required in order to create an application policy, and these abstractions make FBAC particularly well suited to automation. The automation techniques described in this paper can substantially ease the specification of policies.

Utilising FBAC, creating policies for applications using existing functionalities involves assigning functionalities and specifying parameter arguments. In many cases the choice of functionalities is intuitive: for example, the gnobots game is assigned the `Standard_Graphical_Application` functionality, because that is the base-level functionality that describes the type of interface the user expects, and the `Game` high-level functionality, because the user expects the program to act as a game. However, in some cases the choice of functionality, and the values to provide as parameters, can be unclear. For example, selecting platform functionalities (such as `Uses_Perl`) could require knowledge of the frameworks used by the program being confined. Depending on how each parameter is defined, parameter arguments can be file names, directories, ports, or IP addresses. On a Linux system the files and directories that are used by an application are distributed across the filesystem according to the Filesystem Hierarchy Standard (FHS) [16]. Therefore specifying parameter values can require a detailed knowledge of the Linux filesystem. Automating and suggesting the selection of executable paths, functionalities and parameter argument values significantly reduces the burden of specifying application-oriented access control policies.

An LSM-based (Linux security module) implementation of FBAC, known as FBAC-LSM, was extended to implement and test these automation techniques. The automation techniques described in this paper were developed during the analysis of the privilege requirements of existing applications. One hundred and two applications were studied and a set of reusable functionalities were developed. In nearly all the cases studied, functionality and parameter selections can be suggested and automated using the techniques that were developed. Unlike previous approaches to policy specification, complete application policies can typically be specified without having to execute the programs being confined, and with minimal user interaction.

III. Functionality Suggestions

Two techniques have been developed for FBAC-LSM to suggest functionalities during policy construction. Suggestions are based on the libraries that are dynamically linked to the programs or the dependencies of the programs that compose the application, and also the icon category specified for the application. Functionalities are defined using the FBAC Policy Language (FBAC-PL). Definitions include metadata describing the conditions under which each functionality is suggested using the "suggest_functionality" directive. Multiple "suggest_functionality" commands can be defined for a functionality, in which case, if there are matches with any of these, the functionality will be suggested.

When the "suggest_functionality" directive is followed by "uses_library" the subsequent string is searched for within the dynamically linked libraries and dependencies of the application's executables. For example, the following line is included in the definition of the `Audio_Player` functionality:

```
suggest_functionality uses_library "libogg";
```

Libogg is a library used to decode Ogg media files. In this case, when constructing a policy for a new application, if the string "libogg" is found within the linked libraries or dependencies, the `Audio_Player` functionality is suggested.

This analysis can be performed by parsing the output of third party programs. The ldd (List Dynamic Dependencies) command uses the runtime linker to analyse an executable file and generate a list of all the shared objects (.so library files) that are loaded when the program is run. For each executable file associated with an application policy, the ldd command is run by FBAC-LSM with the filename as an argument (i.e. ldd *filename*). If no match for the string is found using ldd, the rpm (RPM Package Manager) command is used to query the RPM database to retrieve a list of the dependencies for the executables. The following command is used to retrieve RPM dependencies:

```
rpm -q -f filename –qf %{NAME} -R
```

The "uses_library" strings that were identified during the analysis of existing applications, and the resulting functionality suggestions are shown in Table I. Any match to the left column results in all the suggestions in the column on the right.

When the "suggest_functionality" directive is followed by "iconcategory", the subsequent string is searched for within the icon categories the application is assigned to. For example, the following line is also included in the definition of the Audio_Player functionality:

```
suggest_functionality iconcategory "Music";
```

On many Linux systems, files with a ".desktop" file extension are used to represent each of the graphical applications

TABLE I: FUNCTIONALITIES SUGGESTED BASED ON LIBRARY USE

Uses_library String (OR)	**Suggested Functionalities (AND)**
libogg	Audio_Player, Audio_Editor
libkmediaplayer	Image_Viewer, Video_Player, Video_Editor
perl	Uses_Perl
Mono	Uses_Mono
Python	Uses_Python
ORBit	Uses_Orbit
java, jpackage,	Uses_Java
xulrunner	Uses_XulRunner
Ruby	Uses_Ruby
kde, gnome, Qt, gtk, X11	Standard_Graphical_Application
libtorrent	BitTorrent_Client
Python-irclib	Irc_Chat_Client

that are installed on a system. These files are used to create icons that can be used to sort and execute these programs. In addition to describing other metadata regarding applications, these files categorise programs. The KDE and Gnome desktop environments use the category to display the application within the appropriate program start-up menus. For example, the Opera application is classified within a file named "opera.desktop" as a "WebBrowser". This information is used to place the program in the appropriate menu in KDE and Gnome. Figure 1 shows how KDE 3.5 uses the icon category to organise launch icons.

The maintainers of software packages typically include '.desktop' files with applications so that they are placed in the appropriate menus. The available icon categories are, to some extent, standardised [17]. The registered categories include these high-level main categories:

> AudioVideo, Audio, Video, Development, Education, Game, Graphics, Network, Office, Settings, System, Utility

Adding additional categories to further specify the purpose of the program is encouraged. Many additional categories are also available.

In many cases these icon categories describe the high-level functions the programs are designed to perform and correlate directly to FBAC-LSM functionalities. The 'iconcategory' strings and subsequent functionality suggestions that were developed are shown in Table II.

Although not a significant problem, functionality suggestions can be subject to some false positives and false negatives. False positive functionality suggestions could occur due to extraneous 'uses_library' substring matches. During the analysis of applications no irrelevant 'uses_library' substring matches occurred, although it is possible that an unrelated library could contain one of the strings used. For example, a library named "monochrome" would contain the string "mono" and the Uses_Mono functionality would therefore be suggested. False positives can also occur when an icon category is more general than the corresponding FBAC functionalities. As shown in Table II, the icon category FileTransfer results in the suggestions: Ftp_Client, Downloader, BitTorrent_Client. At least one of these suggestions would likely be a false positive. False negatives can occur when non-standard or no libraries are used, or when '.desktop' files are not provided.

False positive functionality suggestions are unlikely to pose a significant problem as users must still make the conscious decision to add the suggested functionalities to the application policy. Also, so long as the suggestions contain useful related functionalities, they are helpful. False negatives generally force the user to browse through the functionalities, when creating an application policy, to manually identify the functionalities that describe the features the application provides. FBAC functionalities are assigned functionality-categories that are used to group related functionalities and ease the selection process.

During the construction of policies for the applications analysed, all platform functionalities were reliably and accurately suggested based on the libraries and dependencies of programs. In most cases the icon category-based analysis yielded applicable high-level functionality suggestions.

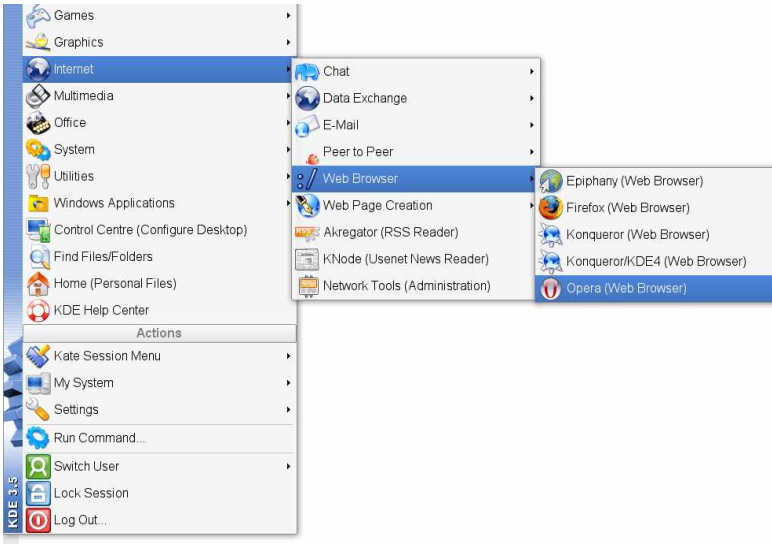

Figure 1: Example of How KDE Uses Icon Categories

TABLE II: FUNCTIONALITIES SUGGESTED BASED ON ICON CATEGORY

Iconcategory String (OR)	Suggested Functionalities (AND)
P2P	BitTorrent_Client
Email	Email_Client
FileTransfer	Ftp_Client, Downloader, BitTorrent_Client
IRCClient	Irc_Chat_Client
TextEditor	File_Viewer, File_Editor
Archiving	Archive_Veiwer, Archive_Editor
Music	Audio_Player
Office;Viewer	Document_Viewer, PDF_Viewer, File_Viewer
Graphics	Image_Viewer, Image_Editor
Player	Video_Player, Audio_Player
AudioVideoEditing	Audio_Editor, Video_Editor
WordProcessor	Document_Editor
WebDevelopment	Web_Files_Editor
Game	Game, Network_Game
WebBrowser	Web_Browser

IV. AUTOMATION OF PARAMETER VALUE SELECTION

Automation techniques can also be used to automate functionality parameter argument selection. Using FBAC-PL, metadata describing automation methods for parameters can be specified within functionality definitions directly after each parameter has been defined. Parameter definitions typically include the default values, description, and type, followed by methods for automating the parameter arguments. Parameter value automation can be performed by searching for matching file and directory paths, and by using the default typical values.

Parameter automation directives start with the command "parameter_automate". If this is followed with "searchforpathmatching" then the policy manager will search for any paths that match the subsequent string. For example, the following line is included in the definition of the *`Standard_Graphical_Application`* functionality for the *peruser_files* parameter:

```
parameter_automate searchforpathmatching
"/home/*/ .**[APPLICATION_NAME]*";
```

As illustrated in the above example, these search strings can contain constants that are represented within square brackets in uppercase. Only one constant has been defined: [APPLICATION_NAME], which represents the name of the current application policy being configured. Constants are only utilised by the FBAC-LSM policy manager and result in literal suggestions that the security module can enforce. That is, if an actual parameter value specified contains the string "[APPLICATION_NAME]", it has no special meaning beyond its literal value, it only has a special meaning to the policy manager for automation purposes.

In FBAC-LSM the wildcard matching for parameter argument automation is performed using a combination of case-insensitive greedy wildcard matching and pattern matching using FBAC-LSM wildcard matching. The result is that case is ignored, and an asterisk (*) matches any character except a slash (/), and a double asterisk (**) matches any character.

The example parameter automation line above therefore tells the policy manager to search the names of all the hidden files in home directories, and within all the files in hidden subdirectories in home directories. Any files that contain the name of the application are added to the values for that parameter. Since the last character in the string is a single asterisk, directory paths will not be added, as they end in a slash. Examples of filenames that would match for an application named "app" are:

/home/cliffe/.apprc
/home/cliffe/.test/app

If the "parameter_automate" directive is followed by "searchfordircontaining" the string following this is searched for within the files contained in directories and, if any files within a directory match the search string, the containing directory is added to the parameter arguments. Pattern matching is performed the same way as described for "searchforpathmatching" commands. These example lines of policy directly follow the definition of the application_libraries_directory parameter for the Standard_Graphical_Application functionality:

```
parameter_automate searchforpathmatching
"/usr/lib/ **[APPLICATION_NAME]*/";
parameter_automate searchfordircontaining
"/opt/
kde3/lib/kde3/*[APPLICATION_NAME]**.so";
parameter_automate searchfordircontaining
"/opt/
kde3/lib/kde3/*[APPLICATION_NAME]**.la";
```

In this case, any directory paths within the /usr/lib directory (including subdirectories) that contain the application name are added. Then, if any library filenames (with the extension .so or .la) are found in the kde3 library directory that contain the application's name, that containing directory is also added.

If the "parameter_automate" directive is followed by "usedefault" then one of two things happens during automation. If this is the only parameter_automate line for the current parameter and the typical values do not include any constants, the parameter is set to use the typical values without any modifications. Otherwise, if there is more than one automate line or constants are used, any constants are replaced with their values and the typical values are added to the list of parameter argument values.

When a parameter has no automation directives, the user is prompted to enter values manually. Otherwise the user is asked to review the values that have been automatically added.

Using the developed functionalities to confine applications, the ability to automatically add parameter values was found to significantly expedite the task of creating policies. During testing, this method of automating values was found to be quite comprehensive, although occasional false positive and false negatives occurred. During parameter argument automation some false positives occurred due to the application name appearing within unrelated paths. This was particularly the case with applications with very short names such as 'vi'.

In the majority of the cases studied, the automated parameter arguments contained all the required values. However, some false negatives were due to non-standard naming of application resources. Although the FHS does not define how application resources should be named, it was found that in the vast majority of cases studied these resource names did conform to unwritten conventions. In most cases directories and files that contain application-specific libraries, configuration files, and other resources contained the command name of the primary executable used to start the application. FBAC-LSM application policies are, by convention, named after the command used to launch the application, and this name is used in the automation process. An example of one of the few applications studied that did not conform to this naming scheme is the Frozen Bubble game, which uses the abbreviation "fb" for its resources rather than the name of the command, which is frozen-bubble. For example, the Frozen Bubble game uses files named ".fbrc" and ".fbhighscores" to store data in users' home directories.

Despite not being standardised, in the applications studied, file naming was consistent enough to facilitate automation of parameter values. From the perspective of policy automation, it would be advantageous to standardise the way application-specific data is named and organised. Standardisation would simplify policy construction for most application-oriented access controls and would improve automation of parameter values for FBAC-LSM. The FHS could be updated to include standard or recommended ways of naming application resources.

Alternatively completely different approaches to filesystem organisation could simplify functionality parameters and value automation. The FHS specifies that the files for a single application are dispersed throughout the system according to the purpose of each file. For example, the executable component is usually stored in /bin or /usr/bin and libraries are usually stored in /usr/lib. The number of parameters required would be reduced if application-specific resources were organised into fewer locations. Organising these files based on the application to which they belong, rather than the type of file, would simplify the parameters and the automation of parameter arguments. The GoboLinux project [18] aims to develop a Linux distribution that takes this type of approach to filesystem hierarchy organisation, where each application has a directory within /Programs/, within which all the files for that program are stored.

A second source of false negatives occurred when applications required access to resources that didn't exist when the policy was created. When an application has not been executed

previously, per-user files will typically not yet exist for that application and these paths are therefore not automatically added to parameter arguments. The FBAC-LSM policy manager's learning mode can be used to add these values to parameters. In the future a real-time access monitor could be developed to detect denied access attempts that match parameter automation strings and accordingly suggest additional parameter values. It is expected that this would completely remove any benefit of running an application unconfined before specifying an access policy.

V. Automation of Executable Paths

Another aspect of policy specification that the policy manager can automate is generating the list of executable files associated with an application. This is achieved using the whereis command to locate the executable files that share the name of the application policy.

A number of the applications studied had additional executable components of the application stored with each application's libraries. For example, when /usr/bin/opera starts, it in turn runs /usr/lib/opera/9.64/opera. Therefore, the policy manager also searches for library directory paths that contain the application name and, if found, it adds an executable path with wildcards, which matches any executable located within the application's library directory.

During program analysis these techniques for automatically selecting executable paths typically resulted in accurate values. False positives seldom resulted from unrelated library paths containing the application policy name, or from the whereis command occasionally returning non-executable files. False negatives could occur if the application policy name does not follow convention, by not matching the application command. Also, occasionally applications are composed of multiple executable files within /usr/bin. This occurred with three of the applications studied: lskat, amarok, and gftp. In the future these techniques could be extended to also look for similar executable names when automating executable path selection.

VI. Discussion

The techniques described in this paper successfully ease the task of specifying policy by making suggestions and by automating policy specification. In the majority of the cases studied these techniques were able to create complete policies for confining applications while allowing them to function legitimately, with very little intervention required by the user. Using automation, the user constructing the policy reviews the paths (which are almost always complete), is asked to select functionalities (the appropriate functionalities are usually suggested), and then reviews the parameter values that have been automatically generated. These values are usually complete, although occasional false positives can be removed by the user and wildcard globing can be used to grant access to related resources.

In most cases complete policies can be created *a priori*, without executing the program. When extra privileges are required, these are usually few in number and can be added using the FBAC-LSM learning tool while enforcing the policy. This is in contrast to policy specification using other schemes, such as Systrace, SELinux, and AppArmor, which typically require those creating policy to perform extensive analysis of the results of learning mode output in order to meaningfully review the detailed policy that is generated. Users are often not qualified to adequately analyse the low-level policies generated by learning modes. Also, executing potentially malicious software while creating policy can pose serious risks as the program is typically not restricted while the security system is learning from its behaviour. Using these other systems it is possible to build policy incrementally while enforcing the policy being created; however, this can be a very tedious task as it can require numerous iterations. To illustrate, one particular participant in a usability study [14] was an IT security professional. He tried to take the more secure iterative and enforced approach using the AppArmor scheme, and found it too tedious and failed to correctly vet the rules that were generated. No tools currently exist for these other security systems to automate complete policy generation without executing the program being confined.

The FBAC model is unique in its suitability for this type of automation. The policy abstractions map to high-level attributes, which can be detected, and the details that need to be specified can be deduced based on the functionalities utilised. These details can in turn also be automated. Future schemes that also provide reusable parameterised policy abstractions could also adopt these automation techniques.

The policy automation approach presented in this paper lowers the expertise required in order to specify application-oriented access control policies, and can enable security-conscious end users or administrators to protect themselves against potentially malicious software [14].

VII. Future Automation Research and Development

It is suggested that these techniques may be used to automate policy specification for other rule-based application-oriented access controls, even those that do not natively provide the reusable abstractions that these techniques rely on. It is proposed that this could be achieved by exporting policy developed using FBAC to other policy languages. FBAC-LSM currently includes an 'export to AppArmor' feature, which is currently in development. This feature aims to allow policies specified using FBAC-LSM automation to be exported to the AppArmor security system.

The techniques for policy automation described here demonstrate the feasibility of the automation of complete application policies. There are many opportunities for further research and development into ways to improve these preliminary automation techniques. Linux package management data could be analysed to automate parameter arguments, utilising the list of files created for each program. Source code analysis could be used to detect likely functionalities. Binary executable analysis is more difficult but could possibly also be used to suggest functionalities. Based on the analysis of the paths detected during parameter automation, the policy manager could perform auto-globing when appropriate. For example, temporary files sometimes contain random strings that could be replaced by wildcards automatically. Also a run-time access monitor could facilitate the addition of denied access when the denied resource path matches one of the automation strings.

VIII. Conclusion

This paper has presented a novel approach for automating the specification of application-oriented access control policies. Unlike previous approaches, using these techniques users can typically create complete policies for applications without having to run them first. These techniques demonstrate and leverage the FBAC model's suitability to policy automation, and the advantages of reusable parameterized policy abstractions. The three tasks involved in specifying an FBAC application policy can all be automated: supplying the executables, functionalities, and parameter arguments. These results indicate the suitability of the FBAC model for application confinement, and present ways that the usability of application-oriented access controls can be improved.

FBAC-LSM, which implements the techniques described in this paper and includes all the policies that have been developed, is free open source software available at: http://schreuders.org/FBAC-LSM

References

[1] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, "Native Client: A Sandbox for Portable, Untrusted x86 Native Code," *Communications of the ACM*, vol. 53, 1, pp. 91-99, 2010.

[2] L. Gong, M. Mueller, H. Prafullchandra, and R. Schemers, "Going Beyond the Sandbox: An Overview of the New Security Architecture in the Java Development Kit 1.2," in *USENIX Symposium on Internet Technologies and Systems* Monterey, CA, USA: Prentice Hall PTR, 1997.

[3] A. Whitaker, M. Shaw, and S. D. Gribble, "Denali: Lightweight Virtual Machines for Distributed and Networked Applications," in *5th USENIX Symposium on Operating Systems Design and Implementation* Boston, MA, USA: USENIX Association, 2002, pp. 195–209.

[4] S. E. Madnick and J. J. Donovan, "Application and Analysis of the Virtual Machine Approach to Information Security," in *ACM Workshop on Virtual Computer Systems* Cambridge, MA, USA: Harvard University, 1973, pp. 210-224.

[5] P.-H. Kamp and R. Watson, "Jails: Confining the Omnipotent Root," in *2nd International System Administration and Networking Conference (SANE 2000)* Maastricht, The Netherlands, 2000.

[6] A. Tucker and D. Comay, "Solaris Zones: Operating System Support for Server Consolidation," in *3rd Virtual Machine Research and Technology Symposium Works-in-Progress* San Jose, CA, USA, 2004.

[7] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer, "A Secure Environment for Untrusted Helper Applications: Confining the Wily Hacker," in *6th USENIX Security Symposium* San Jose, CA, USA: USENIX Association, 1996.

[8] N. Provos, "Improving Host Security with System Call Policies," in *12th USENIX Security Symposium* Washington, DC, USA: USENIX Association, 2002.

[9] C. Cowan, S. Beattie, G. Kroah-Hartman, C. Pu, P. Wagle, and V. Gligor, "SubDomain: Parsimonious Server Security," in *USENIX 14th Systems Administration Conference* New Orleans, LA, USA: USENIX Association, 2000.

[10] P. Loscocco and S. Smalley, "Integrating Flexible Support for Security Policies into the Linux Operating System," in *FREENIX Track: 2001 USENIX Annual Technical Conference* Boston, MA, USA: USENIX Association, 2001, pp. 29-42.

[11] T. Harada, T. Horie, and K. Tanaka, "Task Oriented Management Obviates Your Onus on Linux," in *Linux Conference 2004* Tokyo, Japan, 2004.

[12] Z. C. Schreuders and C. Payne, "Functionality-Based Application Confinement: Parameterised Hierarchical Application Restrictions," in *SECRYPT 2008: International Conference on Security and Cryptography* Porto, Portugal: INSTICC Press, 2008, pp. 72-77.

[13] Z. C. Schreuders and C. Payne, "Reusability of Functionality-Based Application Confinement Policy Abstractions," in *10th International Conference on Information and Communications Security (ICICS 2008)* Birmingham, UK: Springer, 2008, pp. 206-221.

[14] Z. C. Schreuders, T. McGill and C. Payne, " Empowering End Users to Confine Their Own Applications: The Results of a Usability Study Comparing SELinux, AppArmor and FBAC-LSM," unpublished.

[15] Z. C. Schreuders, "FBAC-LSM: Protect Yourself From Your Apps." Accessed 2011: http://schreuders.org/FBAC-LSM

[16] "Filesystem Hierarchy Standard v 2.3," R. Russell, D. Quinlan, and C. Yeoh, Eds., 1994-2004: http://www.pathname.com/fhs/

[17] W. Bastian, F. Gouget, A. Graveley, G. Lebl, H. Pennington, and H. Wendel, "Desktop Menu Specification." Accessed 2011: http://standards.freedesktop.org/menu-spec/latest/

[18] "GoboLinux - the alternative Linux distribution." Accessed 2011: http://gobolinux.org/