The Functionality-based Application Confinement Model

Z. Cliffe Schreuders · Christian Payne · Tanya McGill

Abstract This paper presents the functionality-based application confinement (FBAC) access control model. FBAC is an application-oriented access control model, intended to restrict processes to the behaviour that is authorised by end users, administrators, and processes, in order to limit the damage that can be caused by malicious code, due to software vulnerabilities or malware. FBAC is unique in its ability to limit applications to finely grained access control rules based on high-level easy to understand reusable policy abstractions, its ability to simultaneously enforce application-oriented security goals of administrators, programs, and end users, its ability to perform dynamic activation and de-activation of logically grouped portions of a process's authority, its approach to process invocation history intersection-based privilege propagation, its suitability to policy automation techniques, and in the resulting usability benefits. Central to the model are 'functionalities', hierarchical and parameterised policy abstractions, which can represent features that applications provide; 'confinements', which can model simultaneous enforcement of multiple sets of policies to enforce a diverse range of types of application restrictions; and 'applications', which represent the processes to be confined. The paper defines the model in terms of structure

Z.C. Schreuders (✉)
School of Computing, Creative Technologies and Engineering
Leeds Metropolitan University
HC CA112, Headingley Campus
Leeds, West Yorkshire, LS6 3QS, UK
Tel.: +44 0 11381 28608
E-mail: c.schreuders@leedsmet.ac.uk

C. Payne · T. McGill
School of Information Technology
Murdoch University
Murdoch, Western Australia, 6150, Australia

(which is described in five components) and function, and serves as a culmination of our work thus far, reviewing the evaluation of the model that has been conducted to date.

1 Introduction

Traditional user-oriented approaches to access control do not prevent applications from misusing the privileges of the user-identities they are associated with. Software is typically trusted to act on behalf of local users; however, malware and software vulnerabilities misuse the privileges of users. Application-oriented access control models can limit the damage that applications can cause by restricting access based on what each application is authorised to perform. However, isolation based schemes (such as traditional sandboxes [1, 2], virtual machines [3,4], and containers [5,6]) generally suffer from workflow and redundancy problems, making it hard for applications with different privilege requirements to interact without circumventing the isolation mechanism. Rule-based schemes (such as type enforcement [7], Janus [8], Systrace [9], AppArmor [10], SELinux [11], and TOMOYO [12]) can facilitate finely-grained authorisation to shared resources; however, these approaches typically suffer from policy complexity and usability issues.

In this paper we define a new access control model, functionality-based application confinement (FBAC), which is designed to overcome limitations of previous rule-based application-oriented access control models, including usability issues related to policy complexity,

and provide enforcement of the security goals of system administrators, application developers, and end users. The paper starts by describing the aims of the scheme and the threats it mitigates, then an overview of FBAC is presented. Following that, each of the five components of the FBAC model are described separately and related to previous research and security models. The components combine to form the complete FBAC model, which is illustrated in Section 2.6, Figure 11. The functional aspects of FBAC are also described, including how privileges are propagated across process invocation and how access decisions are made. Substantial efforts have been made to evaluate FBAC, and here we present the results of policy analysis and review the results of a number of publications that explore the practical implications of the model. FBAC is an implementation independent access control model and is designed to have practical benefits when applied to existing systems. Since practical implications are a primary concern, implementation options and implications are also discussed.

1.1 Access Control Aims

As mentioned, the goal of the FBAC model is to provide rule-based application-oriented access controls that overcome limitations of previous schemes. More specifically, the model aims to: provide reusable policy abstractions, provide manageability and usability benefits, simultaneously enforce application restrictions defined by users and administrators, and provide the ability to dynamically deactivate and reactivate portions of policy. The model aims to be compatible with existing applications that are unaware of security mechanisms.

The model presented here is capable of applying mandatory and discretionary controls to enforce the expectations of users and administrators, and is also able to activate or deactivate portions of policy based on the policy abstractions used. Section 4 gives an overview of the evaluation that has been conducted of the FBAC model and its success regarding the other aims: a Linux proof of concept implementation was used to evaluate the ability of the model to confine applications and express reusable policies that are compatible with existing applications, and a comparative usability study assessed the usability of the scheme.

1.2 Threats FBAC Mitigates

In general the threat that FBAC mitigates is that an executed process can act beyond the expected behaviour of the process and thus violate security goals concerning resource usage. Threats that typically lead to processes violating their security expectations include malicious code due to software vulnerabilities and malware. Due to their differing security goals (as described below) separate users and administrators have diverse expectations of executed programs and consider different actions as legitimate. Although existing mechanisms and models typically consider threats from either the perspective of a single user or an administrator, but not both, FBAC is designed to mitigate the threats faced by both users and administrators. What constitutes legitimate usage is therefore defined as behaviour that is expected from all relevant users and administrators.

From a user's perspective, a process using their identity should be acting on their behalf: only accessing the resources necessary to perform the tasks they wish the process to perform. However, the technical expertise of the user may be limited and they may be unqualified to define every action that should be authorised. The threat is that software may act maliciously and beyond these desired actions. The user's security goal in this case is to protect themselves from misbehaving programs by restricting programs to only perform the functions they wish the programs to carry out.

The administrator of a system may face a number of threats depending on their security goals. They may intend to protect users of the system from the threats facing individual users by using a mandatory access control scheme. In this case the threats are similar to those described above: malicious programs may act beyond the expected behaviour of the programs, to a user's detriment.

Administrators may also consider users to be a potential threat to the system, and therefore intend to confine users in terms of the applications they are authorised to run and what they can do using them. In addition, the administrator may wish to apply policies provided by third parties that also define which actions applications are authorised to perform. For example, the administrator may trust the software author to provide a policy that defines how the application is intended to perform, to limit the effectiveness of the exploitation of software vulnerabilities.

1.3 Access Control Model Overview

The FBAC model enforces access control decisions based on the identity of processes, restricting each application to the privileges necessary to carry out its authorised tasks. Users can restrict the programs they execute using discretionary controls and users can also be selectively restricted in application-oriented terms by mandatory controls: for example, rules can specify
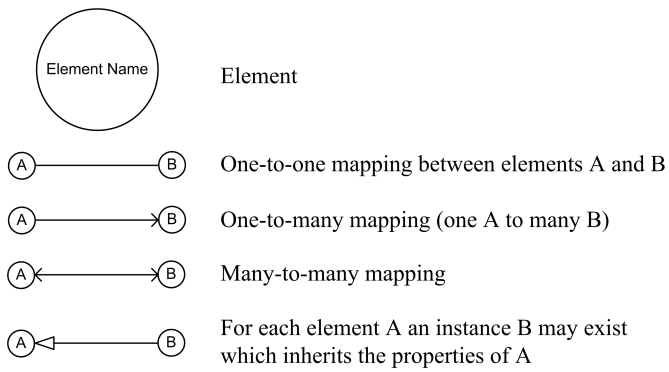
Fig. 1 Model Diagram Notation

which applications users can use and what those applications are allowed to do when particular users are using them. FBAC can be combined with a user-oriented access control model such as traditional DAC or MAC to ensure other user-oriented access goals are also enforced: for example, to specify which resources users are allowed to access.

FBAC restricts applications based on policy abstractions known as functionalities that describe the functions each application provides: for example, web browser, image editor, or email client. Each functionality can be made up of other functionalities in a hierarchical structure. Functionalities are also reusable and can be adjusted through parameterisation to suit the needs of related applications. FBAC can enforce multiple security goals simultaneously by confining users' applications with discretionary and mandatory controls defined by the users or others respectively. Users can dynamically activate or deactivate functionalities of processes. FBAC also allows processes to further confine themselves to the functions they are currently performing. Each of these aspects of the model are described in further detail in subsequent sections.

The FBAC model is divided into five components that combine to form the complete FBAC model: the functionality-based component (which forms the foundation of the model), the hierarchical component, the parameterised component, the user-confinements component, and the process-functionality activation component.

The key shown in Figure 1 defines the notation used in the following model diagrams. The notation employed is similar to that used in the role-based access control (RBAC) standard.

1.4 Relation to Role-based Access Control

Initial work on the FBAC model was motivated by an attempt to leverage benefits from the RBAC model [13] to the specific problem of improving the usability of discretionary system call interposition systems, such as Janus [8] and Systrace [9]. Noting the need for improved usability, manageability and scalability of application-oriented access control models, it was recognised that there is a notional similarity between traditional user-oriented access control (which restricts what users are able to do on a system) and discretionary application confinement (which restricts processes or applications to a subset of a user's privileges). It was contended that the RBAC model mitigated similar problems within the user confinement discipline; that of the policy complexity involved in assigning permissions to users. Therefore, the RBAC model was initially systematically adapted to the context of application confinement by identifying correlations between RBAC elements and restricted execution constructs, adapting RBAC functionality, and by developing a set of operations based on Unix system calls.

Some elements and constructs in the NIST/ANSI INCITS RBAC specification [13] were found to have direct equivalents in relation to application confinement: for instance, applications rather that users access to resources are to be confined. Most importantly, there is a notional similarity between a user's role in an organisation and an application's behavioural class as it pertains to a user's intention for an application: for example, a web browser or an image editor. However, aspects of the model needed further adaptation and careful reconsideration for this new purpose. A major difference between traditional user-oriented access control and application-oriented access controls is that application confinement models such as FBAC need to enable applications to start other applications; bringing with it the complexity of designing effective privilege propagation across process ancestry. Sessions in RBAC define instances of a user within a system, as FBAC confines processes as instances of executing applications; however, there is no equivalent concept in the RBAC model to applications starting other applications, since RBAC does not enable users to establish new sessions owned by other users. Also, as previous research had indicated [16,17], not all applications of a behavioural class require the exact same privileges (unlike most users who have the same roles with RBAC), rather policy can be adapted via parameterisation.

The initial adaptation was therefore followed by redesigns, aiming to extend the model to be parameterised and adaptable to application privilege requirements, to be implementation independent (unlike the first designs which included a set of specific operations to mediate), and support mandatory and discretionary controls. This is in contrast to the initial focus on im-

proving system call interposition to provide a discretionary control; in fact, following the design of the model the implementation that was developed was a system-wide kernel security module, rather than system call interposition. Unlike other application-oriented restriction schemes, FBAC simultaneously provides mandatory and discretionary controls, leveraging the reusability of functionality-based policy to do so with reduced management overhead. These changes in addition to features for dynamically deactivating functionalities required the model to deviate significantly in terms of the RBAC sessions construct. Readers may note visual similarities between diagrams of the FBAC and RBAC models (Figure 11 and [13]), and the distinct differences around the sessions and processes constructs, with FBAC constructs for providing the required application-oriented features. These details are described in Section 2.

By basing the initial design of the new model on RBAC concepts, a so called "functionality-based" scheme has been proposed, which overcomes limitations in existing behaviour based application confinement schemes. Similarities and differences between the RBAC and FBAC models are further discussed throughout this paper, as is this work's relation to other models and literature.

Unlike the role-based access control (RBAC) model standard [13], which separates parts of the model that can be optionally combined when designing a RBAC mechanism, FBAC components depend on each other and are separated primarily to simplify the description and discussion of the model. As described, FBAC attempts to leverage RBAC-like policy abstractions to improve application-oriented access control policy complexity issues. However, it should be noted that the two models differ considerably in terms of goals and context, and in order to meet the stated aims of the access control model FBAC deviates significantly from the structure of the RBAC model.

2 Model Components

2.1 Functionality-based Component

The foundation of the FBAC model is the functionality-based component. FBAC is based on the paradigm of restricting or auditing applications based on the functionalities associated with each application. This component of FBAC is a fundamental aspect of the model and is the basis for all the other components. The term functionality-based is coined, not only to describe this component of the FBAC model, but to also describe any subsequent model based on this paradigm.
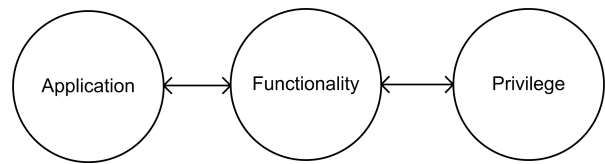

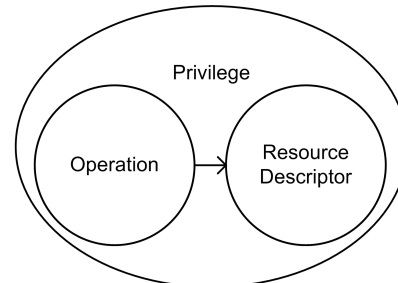**Fig. 2** The Paradigm: Functionality-based


Fig. 3 FBAC Privileges

The functionality construct is designed to represent a function or behaviour an application may be authorised to carry out. Functionalities can describe high level features such as 'web browser', 'email client', 'web server' or 'file manager', or can describe lower level application tasks such as 'HTTP client'.

As illustrated in Figure 2, each application is associated with one or more functionalities in a many-to-many relationship. Functionalities are associated with the privileges required to provide those functions. Privileges are therefore not directly associated with individual applications, but rather via abstract constructs, designed to minimise the management task of assigning privileges to applications.

As illustrated in Figure 3, a privilege in the FBAC model is made up of a single operation associated with one or more resource descriptors. An operation describes the type of access to the resources defined by the resource descriptors.

An FBAC resource descriptor represents mediated resources or, in access control terms, it can represent a set of objects. As shown in Figure 4, a resource can be described by multiple descriptors. Likewise a descriptor can refer to multiple resources.

Although outside the scope of the access control model specification herein, implementations of the model may express descriptors in various ways. For clarification of the concept it is sufficient at this point to simply review some options the model allows. An FBAC resource descriptor may be implemented as label-based or name-based. That is, resource descriptors can be analogous to types in domain and type enforcement (DTE) [14], where one or more resources are labelled with a type that is then used to represent those resources in policy. Alternatively, similar to the way AppArmor
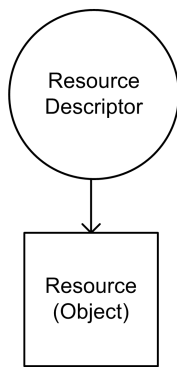
Fig. 4 Resource Descriptors
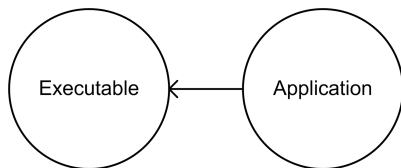
Fig. 5 Executables Associated with an Application

specifies rules [10], the resource can be referred to by a name or string pattern that can identify resources: for example, the pattern "/home/*" can represent any file in the home directory of a Unix system. In either case the resource descriptor can be used to identify resources to the finest granularity of the way resources are distinguished on the system and can also provide some limited abstraction to describe related resources.

An executable file is a stored program that can be executed. Each application has one or more executables (as shown in Figure 5) that are considered part of that application. Processes executing those executables are therefore restricted (as described in subsequent sections) based on the application and the functionalities available to the application.

This concludes the description of the primary component of FBAC. The functionality-based component, which combines the elements discussed in this section, is shown in Figure 6. Each of the elements and relationships illustrated in the diagram has been discussed above. This component describes how, through functionalities, privileges are associated with applications. For the sake of clarity, the way these associations are related to users and processes are described in separate components of the FBAC model. Also, the way the model makes access decisions is covered in detail in later sections.

2.1.1 Discussion of the Functionality-based Component

Other application-oriented restrictions are generally monolithic in nature, where each process has one large detailed policy that applies to it at a time. Rule-based application-oriented access controls that do provide some form of abstraction present in policy, generally do not provide abstractions that are easily reusable for different applications, and they are "compiled" down to a flat list of rules that apply at run-time. In contrast FBAC application policy (that is, the rules that define the authorisation that applies to an application) is made up of multiple reusable abstractions that apply at run-time. The functionality-based component, establishes the foundation of the FBAC model, and describes how application policies are granted privileges via the 'functionality' abstraction.

The functionality-based paradigm (as first proposed in an initial version of the model [15]) was inspired by an attempt to apply the structure of the user-oriented RBAC model to the context of restricting applications. The abstract nature of roles, which form associations between users and privileges in RBAC, led to the concept of applying an abstract relationship between applications and the privileges they require.

This component is comparable to ANSI INCITS/ NIST Core RBAC [13] in that it provides the basis of the model. However, in addition to the different aim of the model, this FBAC component does not include a concept that represents the policy associated with an instance of the subject (represented by sessions in RBAC), as this is left to another FBAC component. The user-confinement FBAC component that provides this aspect takes a very different approach and has a different structure to the one used in RBAC. Also the 'executable' element has no correlation in the RBAC model, as user authentication is outside the scope of RBAC. However, the notionally analogous process-application identification is within the scope of the FBAC model, and as detailed in Section 3 is used for calculating privilege propagation based on administrative FBAC privileges.

The paradigm of confining applications based on the functionalities they perform is related to the concept of behaviour-based sandboxing [16,17]. The main distinguishing feature being that functionality-based restrictions specify multiple functionalities that apply to an application, rather than restricting each program to a single behavioural class. The subsequent components of the FBAC model add substantial improvements to the restrictions provided by this paradigm. As previously mentioned, aspects of the model are separated into components to simplify explanation.

2.2 Hierarchical FBAC Component

The hierarchical FBAC component describes functionality-functionality relationships, where a functionality may
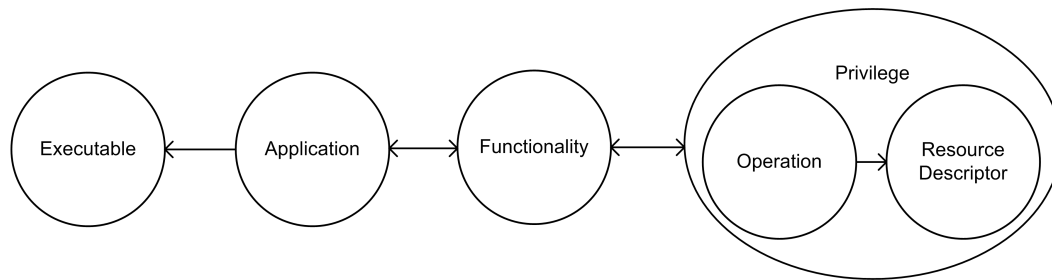
Fig. 6 FBAC Component: Functionality-Based

contain other functionalities. This is shown in Figure 7 as an arrow from the functionality element back to itself.

The hierarchical nature of an FBAC policy allows layers of abstraction and encapsulation to be built. High-level functionalities that describe the purposes of applications (such as Web_Browser, Email_Client and Web_Server) are constructed using lower-level functionalities that provide the authorisation necessary to perform required tasks (such as http_client, ftp_client and POP3_client). These in turn are made up of very low-level abstractions that group finely grained privileges needed to access resources (such as file_r, and file_w).

2.2.1 Discussion of the Hierarchical Component

Unlike other application-oriented models, the hierarchical nature of FBAC policy allows detailed application-oriented policies to be formed from layers of policy abstractions. Policy abstractions are typically self-contained, have limited reusability or adaptability, and are compiled into a single set of rules that are applied at run-time. This level of abstraction and reuse of policy has not been incorporated into the access control model of previous application-oriented schemes. Some schemes, such AppArmor, have policy languages that can convert from simple hierarchically contained abstractions to a flat list of rules to be enforced. However, unlike these schemes, FBAC can make access decisions based on these hierarchies, which means they can be activated or deactivated dynamically, as formalised in the process-functionality activation component.

The hierarchical FBAC component is similar in concept to the hierarchical RBAC component in its "general hierarchies" form. General RBAC hierarchies also allow multiple inheritance/containment of policy abstractions. FBAC hierarchies are distinct in that they use containment, where a functionality contains another in terms of privileges, and contained functionalities can be deactivated (the method of activating and deactivating functionalities is described later in Section 2.5). Whereas RBAC hierarchies can only be deactivated

from the highest level roles that are associated with users, contained roles cannot typically be deactivated individually. For this reason it is possible to describe RBAC hierarchies as inheritance, since all the attributes of inherited roles are effectively transferred to the parent roles, whereas FBAC hierarchies are contained rather than inherited. This is a deviation from the RBAC structure so as to allow greater run-time control over the functionality hierarchy.
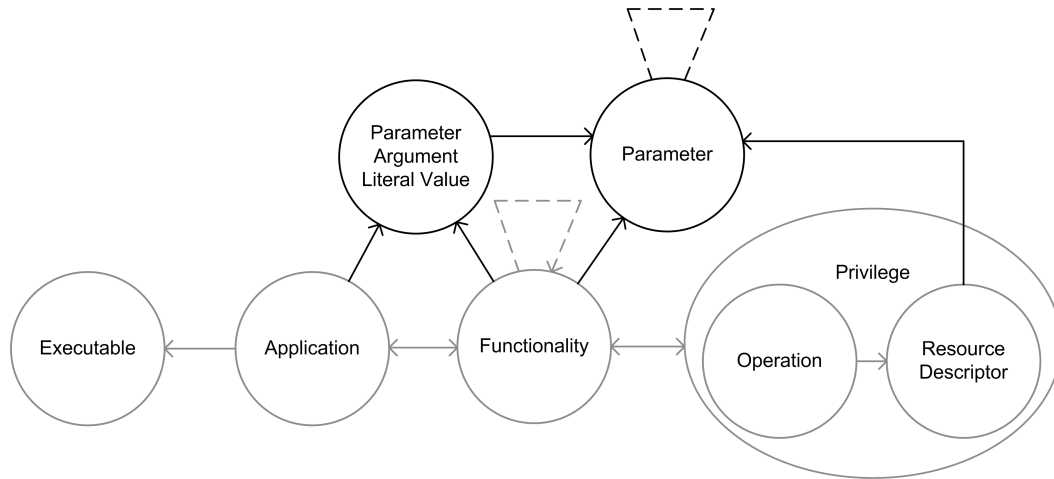
2.3 Parameterised FBAC Component

FBAC is parameterised, allowing functionalities to adjust the resource descriptors of contained privileges to adapt to application specific requirements and therefore allow access to resources required by particular applications. As illustrated in Figure 8, functionalities can have multiple parameters. Application policies can then send arguments to those parameters when functionalities are assigned. Parameter arguments hold literal values, which are specified in an application policy or functionality. These values can be assigned to a parameter of a functionality from within an application policy or to a contained functionality within a parent functionally. Functionalities can then use parameters in place of literal values as resource descriptors. Functionalities can therefore grant access to resources that can be defined when the functionality is assigned, allowing functionalities to be reused to grant access to different resources as needed.

For example, a functionality named Standard_Graphical_Application could have a parameter called peruser_directory. The application policy for a program such as Firefox would then have a literal value describing the resource requirements of that application (such as "/home/*/.mozilla/firefox/", on a Unix-based system), which it sends to Standard_Graphical_Application when it is associated with the application. Standard_Graphical_Application can then grant access to the parameter peruser_directory using a privilege that uses the parameter as a resource descriptor. Other functionalities such as Web_Browser would also be used and sent
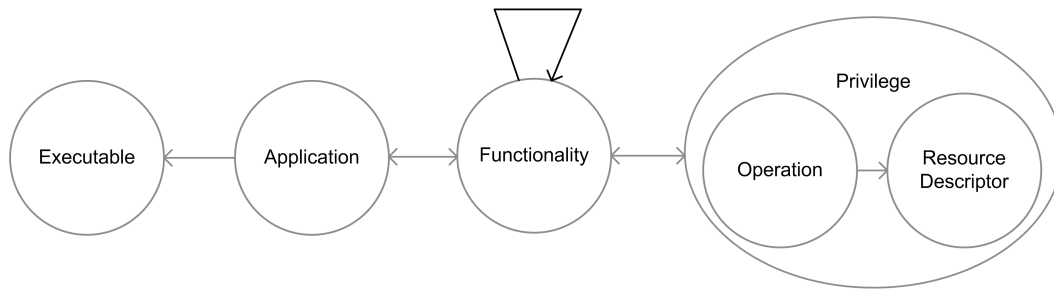
Fig. 7 FBAC Component: Hierarchical FBAC

Fig. 8 FBAC Component: Parameterised FBAC

parameter arguments to customise these functionalities to the application.

When functionality hierarchies are present (where functionalities contain other functionalities), then parameters also form hierarchical structures. This is shown in Figure 8 as dashed lines. Functionalities that contain other functionalities can send their parameters as arguments to contained functionalities' parameters. So in the example in Figure 8 the Standard_Graphical_Application functionality can use another functionality such as dir_full_access to grant the necessary access by passing the parameter peruser_directory as a parameter argument to the contained functionality dir_full_access. These hierarchical functionality levels encapsulate details while providing flexible abstractions that can be fine-tuned to suit the diverse implementation details of related applications.

FBAC functionalities are therefore passed arguments in a fashion similar to subroutines in programming languages. This allows the policy abstraction to easily adapt to the differing details of applications providing related features. Functionalities contain other parameterised functionalities and parameterised privileges; where privileges are computed at run-time based on the arguments passed to privileges via functionalities. This hierarchical relationship between functionalities allows arguments to propagate to any contained functionality.

2.3.1 Discussion of the Parameterised Component

Previous research has demonstrated that the resource needs of programs can be related to behavioural classes and that applications can be restricted with some success based on the class of program along with parameters that describe the specific needs of a program [17]. FBAC combines this general approach with the functionality-based scheme described in the previous sections to provide a model to restrict a process using multiple behavioural classes. Hierarchies improve policy by abstracting details, while these abstractions are themselves parameterised to allow them to also adapt to the specific needs of the situation.

While in RBAC it is usually adequate for all users in a specific role to have access to the exact same resources, it is not sufficient for all applications performing the same function to have access to the exact same resources. For example, applications typically store their configuration files in separate directories. The addition of parameterisation is therefore necessary in order to apply an RBAC-like structure to the context of behaviour-based restrictions. A few RBAC schemes have been

proposed that incorporate some form of role parameterisation, such as those proposed by Giuri and Iglio [18] and by Yao et al. [19]. However, these schemes focus on using environmental constraints and object contents to limit the availability of privileges or roles to users. FBAC takes a new approach that is designed for the specific needs of application confinement, where abstractions can be used to grant different privileges based on the needs of applications, and can be layered to encapsulate details so that applications can be confined based on their high level features, which are, in turn, modelled by parameterised lower level features.

Parameterisation significantly changes the semantics of the restriction and resolution of access decisions, and as such differentiates the RBAC and FBAC models substantially. RBAC roles are normally omnidirectional in the sense that it is equally demanding to determine which users are assigned to a role and what permissions a role grants [20]. On the other hand, FBAC functionalities are unidirectional in the sense that it is easier to determine which functionalities are available to applications than to determine what privileges a functionality will grant. This design enables FBAC functionalities to be adjusted to the specific needs of applications, whereas RBAC roles generally only provide the exact same privileges to each user assigned.

This component has some notional similarities with subsequent work on access control policy templates by Johnson et al. [21], where policy templates/abstractions are developed separately from policy specification. However, the models, methods, and aims of the schemes are distinct. The scheme proposed by Johnson et al. aims to assist in generating domain specific structured lists for guided policy authoring, where as FBAC focuses on providing usable application-oriented controls and takes a different approach to policy development and specification.

2.4 FBAC User-Confinements Component

The FBAC model is capable of simultaneously enforcing multiple application-oriented policies that apply to specific users. This is achieved by the FBAC User-Confinements Component described in this section.

Using FBAC, mandatory controls can restrict users in terms of the applications they are allowed to execute and what those applications can subsequently do. Furthermore, by allowing users to have discretion over some policies pertaining to their own applications, the model can also enforce discretionary controls that users can utilise to restrict their own processes.

An FBAC confinement represents application restrictions that apply to specific users. As illustrated in Figure 9, each user can have multiple confinements that apply to them (shown as the right-side connection between User and Confinement). Each confinement also has users who are authorised to maintain the application policies, which involves application specification and association with functionalities. This is represented on the diagram by the left-side connection between confinements and users.

Each confinement has a list of applications that use functionalities available to the confinement. That is, the applications associated with a confinement can only be associated with functionalities from that confinement. Due to the reusability of functionalities, functionalities can be made available to multiple confinements and hence the many-to-many relationship between the functionality and confinement element.

As shown in the dashed rectangle in Figure 9, each confinement also has an attribute value applies_to_type; this allows the confinement to apply to all the users associated with the confinement (the value only), to all other users (except), or to all users (everyone). Each confinement also allows configuration of the policy that is to apply to processes that do not have a matching executable. This is specified using the no_profile value, which can take on of the following values: unconfined, confine_with_restricted_profile, or deny_execution. In the unconfined case a child process is restricted by its parent's policy.

Creating mandatory restrictions involves creating confinements that apply to users that are not also maintainers of those confinements. In terms of Figure 9, there is an 'applies to' relationship between a User and a Confinement, but not a 'maintained by' relationship. Usually mandatory confinements would be maintained by a security administrator, although other less likely configurations are possible such as allowing one normal user to specify the application restrictions of another user (such as for a colleague with less technical knowledge). Creating a discretionary control involves creating a confinement that applies to a user who is also authorised to maintain the confinement. The user can then add applications to that confinement to restrict programs.

2.4.1 Discussion of the User-Confinements Component

The FBAC user-confinements component describes a significant aspect of the FBAC model that is unique within the field of application-oriented access control: the simultaneous enforcement of multiple policies defined by multiple people with distinct security goals applying to the same process. The applies_to_type and the no_profile confinement attributes enable the admin-
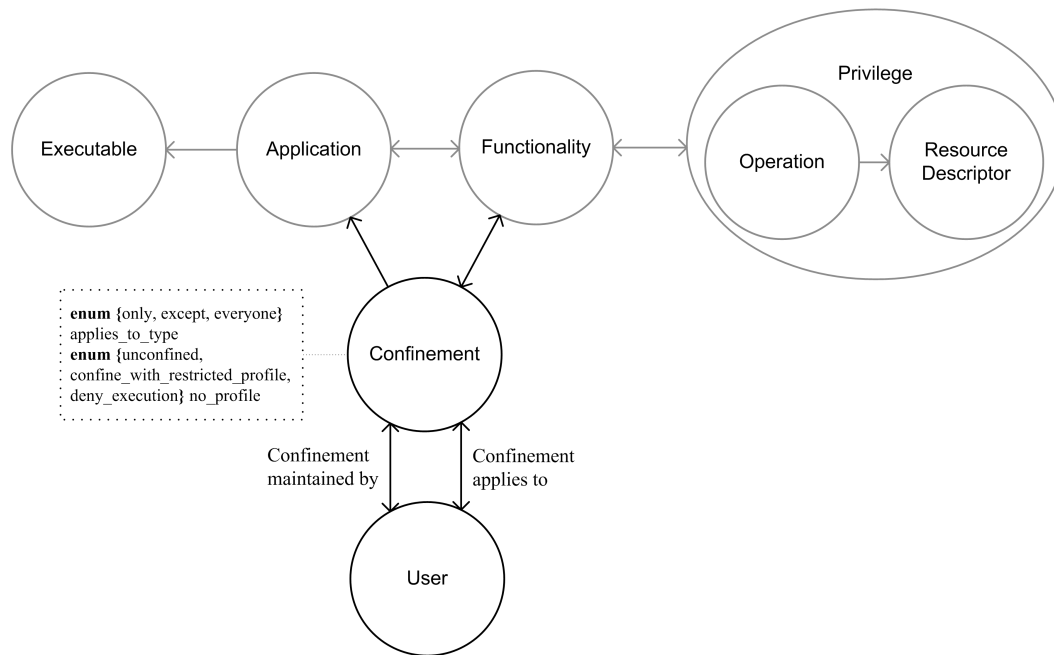
Fig. 9 FBAC Component: User-Confinements

istrator to configure a number of different types of restrictions. For example, sets of users can be confined to using only particular programs. Other programs can either be off limits (deny_execution), or severely restricted (confine_with_restricted_profile). Alternatively a more targeted approach can also be achieved, where programs without policies are unconfined. The 'maintained by' relationship between users and confinements allows administrators to authorise users to configure the rules for applications to also enforce their own security goals. These constructs can be utilised to enforce a number of diverse security policies, and can represent rules not easily enforced using previous schemes.

Previous application-oriented models were designed to enforce one policy, which is specified to enforce a particular security goal, for each process or program. Mechanisms are therefore designed to either provide a mandatory or a discretionary control. For example, systems such as Janus [22], Systrace [9], and TRON [23] provide enforcement of discretionary user-defined process restrictions, and systems such as AppArmor [10], SELinux [11], and Linux DTE [24] are system-wide mandatory controls defined by an administrator. These systems are therefore used to enforce separate goals. Discretionary application-oriented controls are employed by users to protect their own resources from a malicious program. This ensures the program only accesses resources required to carry out the tasks the user wants it to. In contrast, mandatory controls are used by administrators to enforce system-wide security goals, ensuring that processes do not access resources

that could lead to these goals being subverted: for example, the restriction of certain shared services such as web, ftp or local setuid programs. Both approaches have security benefits. However, providing both types of restrictions using previous models requires two separate mechanisms to be maintained. Even if the same model (for example, DTE) was implemented as both a mandatory and discretionary control, it would involve the redundancy of maintaining both types of controls entirely separately.

On the other hand, FBAC is designed to provide both types of controls, and does so while reducing the overhead of enforcing multiple policies for a single process. The main policy unit (functionality) is reusable across confinements, which makes the task of maintaining low-level policy scale well to this situation. Users and administrators can reuse these abstractions in any of the confinements they maintain to enforce their own security goals. Enforcement is achieved through the one model in one access decision procedure.

Most application-oriented models and mechanisms do not consider the user identity when confining a process. Discretionary application-oriented controls typically only apply to a single user maintaining the restriction, while mandatory application-oriented controls typically apply the same set of rules for a program regardless of user-identity. The main exception to this is SELinux [11], which is a framework that combines multiple security models, including non-standard versions of RBAC and DTE. Roles define which domains users are authorised to transition into. SELinux policy

takes the form of an extremely complex combination of a number of security models, which makes it hard to maintain and verify for correctness [25]. FBAC provides user-specific restrictions using a single application-oriented access control model that is designed to be easier to administer.

Because the FBAC model only involves positive authorisation rules (not denials), conflicts in security goals do not result in decision making complexity. Rather, if a user has the authority to maintain a confinement, then they may configure policies that can further restrict the actions of any programs run by the users that the confinement applies to. The model can enable clear auditing to make the source of denials clear. Although not a common use case, assigning multiple users to maintain a single confinement should involve careful consideration, to ensure they collaborate on policy effectively, since they will all have the authority to modify any rules that applies to that confinement.

2.5 FBAC Process-Functionality Activation Component

Figure 10 shows the FBAC process-functionality activation component. Included in this figure are the process and task confinement elements, which associate policy with a running process. As shown in the figure, a user can run multiple processes. Each process is confined by all the task confinements that correspond (one-to-one) to any confinements that apply to the user. For example, if a user is restricted by two confinements (such as a mandatory and a discretionary confinement), then any process owned by that user would have two active corresponding task confinements. That is, each confinement that applies to a user is instantiated as a task confinement for each of their processes, and inherits the values of the confinement (applies_to_type and no_profile). If the process corresponds to an executable that matches an application that is associated with one of its task confinements, then the task confinement has a one-to-one association with the corresponding application (shown in the figure by the connection between Task Confinement and Application).

Confinement process ancestry is represented by mapping task confinements to task confinements (children to parents). This relationship is shown in Figure 10 as the connection between the Task Confinement element and itself. This parent-child relationship is used to calculate privilege propagation and used in access control decision logic, as described in subsequent sections.

For each task confinement, each of the functionalities associated with the application are instantiated as functionality instances. The associations between task confinements and functionality instances represent which functionalities are active for a task confinement. By default, when a process is started, all its functionalities for each task confinement that apply to it are activated. However, this behaviour could be altered in the application's policy to require manual user activation. Functionalities directly assigned to applications are activated by mapping functionality instances to task confinements (labelled in Figure 10 as "Active high-level functionalities"). Inherited functionalities are activated by mapping functionality instances to other functionality instances (labelled as "Active hierarchical functionalities"). By default, when a functionality is activated, all inherited functionalities are also activated. Functionalities can be deactivated by severing these relationships, and reactivated by re-instantiating functionalities. Processes are only granted access to privileges via functionalities that are active.

The functionalities associated with a task confinement can be dynamically activated and deactivated. Users who have discretion over policy (that is, they have a 'maintained by' relationship with the confinement, as described in the FBAC User-Confinements component) can activate or deactivate them, while security aware software (i.e. the process itself) can only deactivate functionalities. This allows the rights of a process to be altered at run time by dynamic interaction. Users can limit the software to the behaviour the user wants the application to carry out at a point in time, while software can make itself more resistant to vulnerabilities by restricting itself to only the functionalities it is currently performing.

Users who have discretion over policy can deactivate functionalities associated with the corresponding task confinements. The functionalities may then be reactivated only by further user intervention. Processes may also deactivate functionalities if they are FBAC-aware. Although a process can drop functionalities, it can never reactivate them. Software authors can therefore limit the impact of vulnerabilities by dropping all functionalities other than those that represent the task a process is currently performing. For example, a program that can both act as an email client and a web browser could fork a process for performing web browsing and then deactivate the email client functionality. This behaviour would limit the impact of malicious code to within the permissions associated with web browsing and would not allow the browser to send emails if the browser component of the software was compromised.
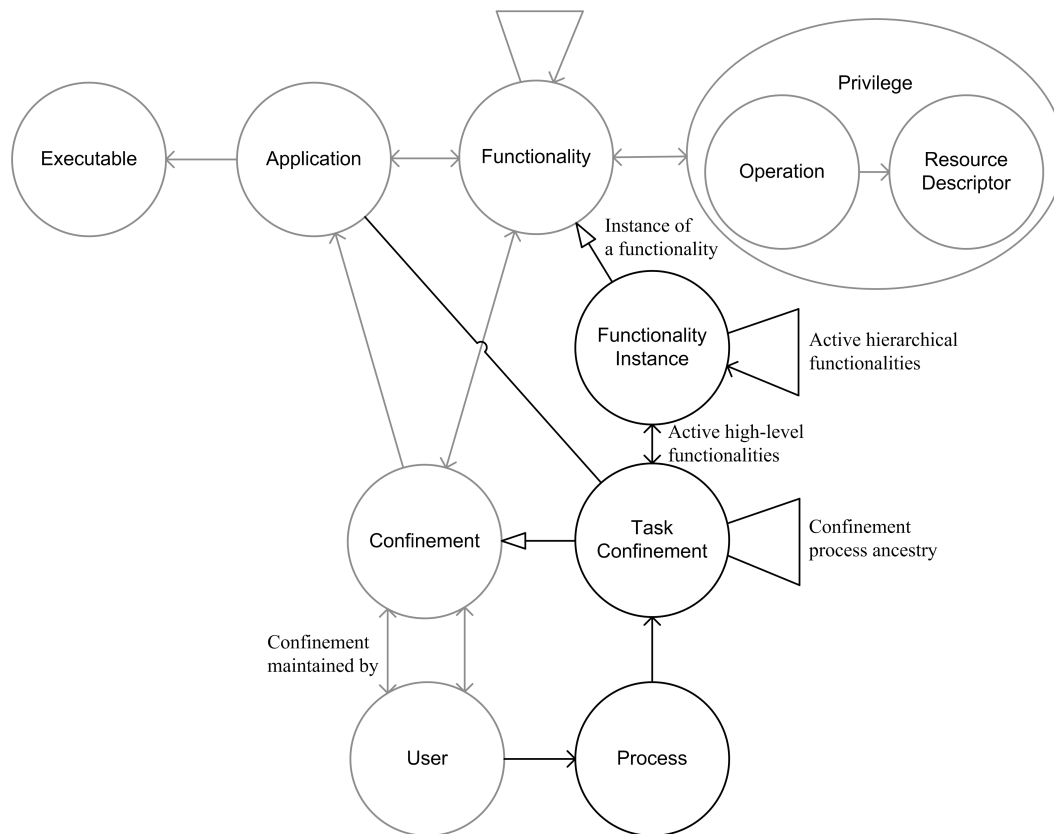
Fig. 10 FBAC Component: Process-Functionality Activation

2.5.1 Discussion of the Process-Functionality Activation Component

By allowing processes to further restrict themselves, FBAC enforces yet another type of restriction, application security, without introducing any management overhead to users or administrators. This restriction is another form of discretionary control, but from the perspective of the processes rather than the user. Other such restriction schemes include FreeBSD Jails, and chroot() [5], both of which allow a process to initiate one-way privilege declination. FBAC provides the functionality abstraction to processes, which makes managing privileges much simpler for the process than managing each privilege independently. FBAC also provides much greater control over privileges than namespace scope-limiting schemes such as Jails and chroot()or coarsely grained schemes such as Linux capabilities [26] or the Mac OS X sandbox API [27, pp. 156-178].

FBAC's hierarchy of functionalities allows run-time intervention to dynamically deactivate or activate branches of functionalities. This is similar to the concept of users restricted by an RBAC scheme who only activate the roles relevant to the part of their job they are currently performing in order to mitigate the security risks involved in holding excess privileges.

Although related to the idea of active roles in RBAC, the scheme for providing active functionalities in FBAC is distinctly different to the structure used by RBAC. RBAC uses the concept of sessions, a simple mapping between users and the roles they have activated [28]. Rather than simply providing a mapping between task confinements and functionalities, FBAC introduces the concept of functionality instances, which allows functionalities to be dropped or activated from within a hierarchy. This level of dynamic control is not possible using RBAC. So although the FBAC model was developed in part from the RBAC model, FBAC allows greater dynamic control of policy than the RBAC model allows. For example, a 'Web Browser' functionality could contain other functionalities that allow HTTP and FTP network access. Using FBAC, a web browser process could drop the ability to use FTP while still using HTTP. Using RBAC, only roles directly associated with a user can be activated, and any inherited roles are automatically also active.

Because FBAC's policy abstractions are hierarchical, small or large parts of the policy can be activated or deactivated at run time. This is not possible using ex-

isting application-oriented access control models, such as DTE [29]/SELinux [11], RC [30], or AppArmor [10], as privileges are contained in a monolithic abstraction associated with the security context. Changing privileges using these models requires transitioning into another complete set of privileges (domain, role, or profile respectively). FBAC provides greater dynamic control of active policy than any previous application-oriented access control.

2.6 Complete FBAC Model Structure

The FBAC model as a whole is made up of the previously described components and is shown in Figure 11. The following example demonstrates how the FBAC model elements relate to each other. The example also refers to the decision making process that the element relationships are used for; this aspect of the model is described more formally in the Appendix.

The user Alice may have two confinements that apply to her: one that she maintains called "Alice's_Discretionary", and one that her system administrator put in place called "Staff_Mandatory", which is non-discretionary and applies to all staff. These user-confinement relationships — the confinements that apply and who manages them — are shown as the two arrows between Confinement and User.

Alice and her administrator may have both created policies for the program Firefox. The application policies specify which executable files form part of the application (the arrow from Application to Executable) and which functionalities the program performs (the arrow to Functionality). By specifying parameters for functionalities (the arrow to Parameter Argument, which is for a particular Parameter), the administrator ensures that the program will only write to Alice's home directory: so that if the program is exposed to malicious code it cannot alter other shared resources Alice has access to. Alice restricts the application further, granting access to particular directories within her home directory, such as a download directory and the applications configuration directory.

When Alice executes the program, its path matches an executable specified in application policies in both confinements and a process is created and the program starts. Two task confinements that correspond to the confinements that apply to Alice are created for that process. Each task confinement links to the application that it is confining the process as (the line between Task Confinement and Application). These task confinements represent the restrictions that are enforced for the process. The functionalities that apply to the application are associated with the task confinement

when those functionalities are active. If a functionality is deactivated, it is removed from this relationship and is no longer used to calculate what that process can do.

When that process attempts to access a mediated resource (for example, it tries to write a file to disk), each task confinement is queried. Each task confinement authorises the action based on the functionalities that are active and the process's ancestry. If every task confinement allows the action then the request is allowed, otherwise it is rejected.

3 Process Ancestry and Authority Propagation

The FBAC model controls authority propagation between processes based on process ancestry and associated privileges. The FBAC model does not allow processes to discretionarily delegate privileges to arbitrary processes.

Each task confinement that applies to a process restricts propagation independently and, for an action to be allowed, every task confinement must permit the action. This section describes the propagation of authority within a single task confinement across processes. The way in which task confinements are combined to form the final authorisations for applications is described in the Appendix.

The FBAC model does not specify a complete list of operations, as operations can be implementation-dependent so that implementations can take advantage of the granularity of the available security mediation interface. However, the FBAC model does specify a small number of operations, presented here, for controlling authority propagation. These operations are considered necessary for the model as they are involved in the decision logic for controlling authority based on process ancestry.

In order for a process (A) to execute another (B) it must have a privilege that explicitly allows this. An execute privilege is specified using one of the operations shown in Table 1, along with a resource descriptor that specifies the programs that can be executed. The amount of authority granted to the executed program B depends on the operation used to grant the permission.

Table 1 gives an overview of the operations that control authority propagation. The prefix "file_" simply denotes that the operations work on resources that are files, while those prefixed with "application_" operate on applications, allowing execution of any file that is an executable of the specified application. The resource descriptors used with any of these operations form privileges that specify the executable files or the other application that the application is authorised to execute.
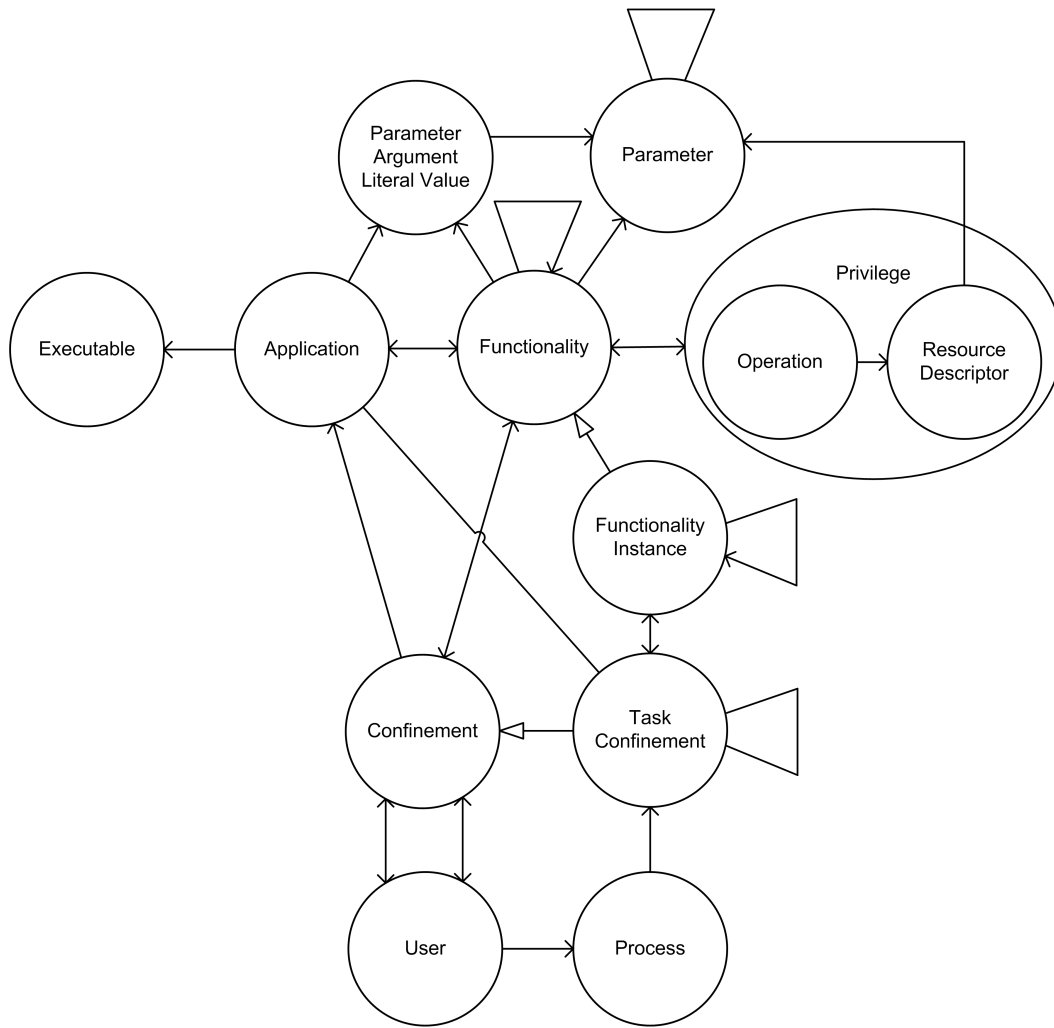
Fig. 11 The Complete FBAC Model

Table 1 FBAC Authority Propagation and Execute Operations

Operations	Description	Propagation of Authority When Application A Executes Application B
$file\_execute$ $application\_execute$	Where an application A executes another application B to help perform the functionalities the application performs. For example, using helper programs to perform subtasks.	Subtractive propagation: $A \cap B$
$file\_execute\_load\_profile$ $application\_execute\_load\_profile$	When application A executes another application B to perform other functionalities that A cannot perform.	Context transition: B
$file\_execute\_shell$ $application\_execute\_shell$	When application A executes a shell such as bash to execute other programs to help perform the functionalities the application performs.	Context copy: A Special case: the next propagation is always subtractive.
$file\_execute\_as\_current\_app$	Can be used when no profile exists for program B to allow B to run as A. This is equivalent to adding executable B to application A, except that executable B is only considered part of application A when executed by application A.	Context copy: A

The way process task confinement privileges are combined is given in the right hand column, where A is the set of privileges that apply to the previous process task confinement. For example, if Firefox, a web browser, starts with its full application permissions (via *_execute_load_profile) and uses rm, which deletes files on Unix, (via *_execute) to remove a file, then rm has the permissions, $Firefox \cap rm$. In the unlikely scenario that rm was authorised to run mv (via *_execute) then the previous permissions A ($Firefox \cap rm$) would be intersected with mv, giving $Firefox \cap rm \cap mv$. The result in this example is that Firefox can use rm to remove files that Firefox is authorised to remove, and rm could not use other programs (via *_execute) to do anything but remove files that Firefox is authorised to remove.

The *_execute operations are used for the frequent situation where an application A executes another application B to help it carry out its features. In other words the subsequent application B acts within the bounds of what the first application A is allowed to do, but since the application B exists, application A uses it rather than reprogramming the task involved. Application B is restricted using subtractive propagation, where application B is restricted to an intersection of the permissions allowed to application A (the result of previous propagation) and the policy allocated to application B. This propagation is safe and always further restrictive, never permissive. This allows an application policy for a program such as rm to allow all file unlinks. However, when it is used by another application to remove files, it can only remove files that application could have removed itself.

The *_execute_load_profile operations are used when an application needs to start a dissimilar application, that is, an application whose resource needs are distinctly different. Application B is allowed the full permissions afforded to the corresponding application policy. This is roughly analogous to allowing a domain transition in DTE. There is an inherent security risk involved as propagation is not necessarily restrictive: propagation can grant program B authority that program A does not have. Therefore these operations should be used with caution and with well thought-out security goals. Examples of when these operations would be used are for a launcher program, which starts applications with their full policies, and for a web browser that is allowed to start a program such as a word processor to view downloaded files. In each case the interactions authorised between applications using these operations need to be carefully considered.

The file_execute_as_current_app operation restricts the subsequent application B to the same security context as the parent application A. This is similar to adding an executable to an application, except that the executable file B is only considered as part of application A when an executable from application A runs it.

The *_execute_shell operations are used for the special case of an application launching a shell through which other programs are executed to carry out the tasks of the first application. This is a common occurrence on Unix systems as programs often start other helper programs via the bash shell. The shell is restricted using the policy of application A, similar to the behaviour resulting from the file_execute_as_current_app operation. However, a special condition applies: when the shell starts another application, propagation is always restrictive. That is, in this case *_execute_load_profile is treated as *_execute. This design allows the policy for a shell to authorise full profile loading of applications when run as a launcher via *_execute_load_profile, and when used by other applications with *_execute_shell it acts as a helper, which is a safe and restrictive approach.

Figure 12 illustrates some example process ancestries (including the Firefox rm mv example above) and the resulting authorised permissions.

When granted multiple privileges to execute the same program, the following order of predominance applies.

1. file_execute_as_current_app
2. file_execute_shell or application_execute_shell
3. file_execute_load_profile or application_execute_load_profile
4. file_execute or application_execute

This ordering allows special cases (with higher privilege) to overwrite general cases. For example, on a Unix system using name-based resource descriptors, an application may be allowed to run any program in "/bin/*" with file_execute, except in the special case of "/bin/bash", which is run with file_execute_shell. This order is designed to simplify policy specification as special cases override common cases. Otherwise in the example above the file_execute operation could not be used as a blanket permission with "/bin/*", instead every separate executable would have to be specified separately because the "/bin/bash" special case would otherwise be overridden. Similarly the file_execute_as_current_app and *_execute_load_profile operations are special cases, which are to be used seldom and with careful consideration.

On modern systems, programs are not always in the form of natively executable files. Scripted and interpreted languages such as bash scripts, Perl and Python, and frameworks such as Java and .NET often have a process interpreting and working on behalf of a separately stored program. For this reason FBAC includes
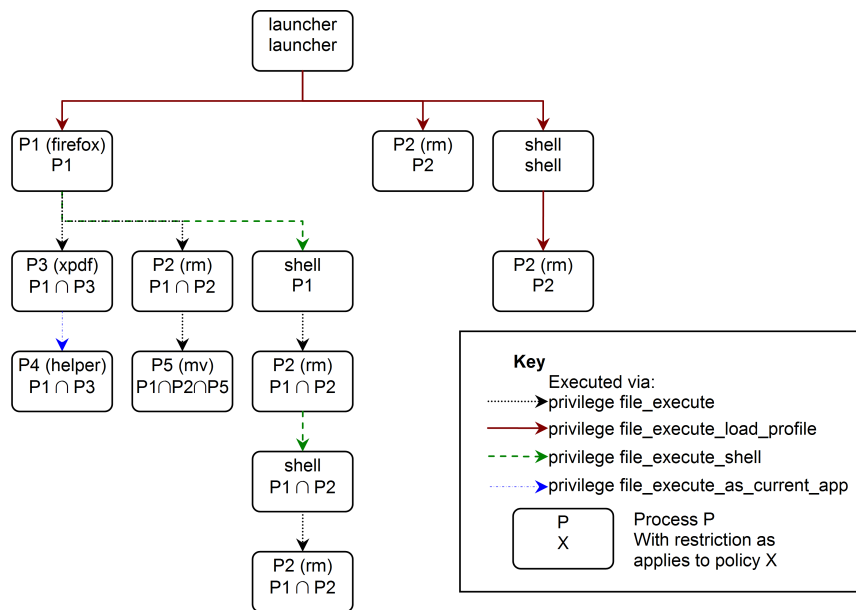
Fig. 12 Hypothetical Authority Propagation Example

an additional operation that grants a program the right to work on behalf of another. The interpreter (A) can then assume the privileges of an application (B) in one of two ways: the interpreter can signal to the access control mechanism that it wishes to do so, or the interpreter can access the file with execute permission. Doing the latter automatically initiates the procedure. The resulting privilege is still subject to the privilege propagation rules previously described.

As shown in Table 2, the privilege granted to the interpreter's application policy is the operation file_execute_as_interpreted or application_execute_as_interpreted with a resource descriptor that defines the file or application that is to be interpreted.

For example, Java may be given the permission to interpret all .class files. An rm.class Java program that deletes files could be created and that class file could be added to the executables of the rm application policy. Then, when Java accessed the rm.class file, Java would assume the privileges of the rm application and be allowed to delete files. However, the process ancestry that led to the execution of Java would restrict which files Java could delete. Returning to the Firefox example, if Firefox executed Java via *_execute, then Java could only access the .class files that Firefox could execute. When Java executes rm.class its privileges would become $Java \cup rm$, which when combined with the fact Java was started by Firefox, becomes $Firefox \cap (Java \cup rm)$. Consequently the interpreted rm program may still only delete files that Firefox can delete. Note that the Linux kernel is aware of certain interpreters, and is therefore able to identify a pro-

cess in terms of the file being interpreted. Therefore, the FBAC-LSM implementation avoids having to implement this part of the FBAC model, and instead assigns additional functionalities to interpreted programs, which enable the interpreters to function.

3.0.1 Discussion of FBAC Privilege Propagation

The FBAC scheme for privilege propagation and process ancestry is unique within the field of application-oriented access controls. Isolation-based schemes typically do not perform security context changes and consequently all children are isolated to the same resources. Rule-based controls typically allow different applications to be restricted using separate policies that are applied when they are executed. When processes start, most rule-based application-oriented access control schemes (for example, AppArmor, SELinux, and DTE) consult the active policy and either keep the parent's security context for the new process or transition to being confined by a separate policy. Unfortunately, with discrete policies it is difficult to verify that all the authorised policy transitions are safe [31]. Transitions can lead to the 'confused deputy' problem, where a program can launch and influence another more privileged program in order to exceed its own authority [32]. The FBAC model simplifies the security sensitivity of many of these interactions by preferring an intersection approach, where each helper process (such as one of the common Unix commands) is confined to the intersection of its own authority and that of the program(s) it is acting on the behalf of. In this case it is safe to allow

Table 2 FBAC Privilege Interpreter Operation

Operations	Description	Resulting Privilege When Application A Interprets Application B
$file\_execute\_as\_interpreted$ $application\_execute\_as\_interpreted$	Allows application A to interpret and act using to privileges of application B.	Additive: $A \cup B$

the parent processes to influence the child process, and the child can be confined to specific behaviour, rather than allowing it the excess authority of either the parent or child policy. This concept was illustrated by the Firefox/rm example in the previous section.

The FBAC privilege propagation approach is related to stack inspection, a language-based security feature of Java and .NET. Stack inspection is used in the context of application virtualisation to restrict the actions of untrusted code modules within a virtual machine [33, 34]. Access rights are the intersection of the authority of all the frames on the stack. Therefore a malicious module cannot perform operations it is not authorised to. Trusted code can optionally assert responsibility for use of some permissions, thus overriding the inspection of its callers: for example, to grant additional privileges [35]. Stack tracing has some notional similarities with the FBAC privilege propagation scheme. In both schemes intersection is performed by inspecting the invocation history; in the case of stack inspection this is the method call history, for FBAC it is the process ancestry history. Also in both schemes intersection is overridden when necessary; for stack inspection this involves enabling privileges, for FBAC this involves making execute_as_current_app transitions. Although the work is related, the specifics and purpose are distinct.

The Singularity operating system incorporates process invocation history in its system-wide access control scheme [36]. Process identities are represented using text strings, which include the complete process ancestry, and can be used in access decisions to confine applications by pattern matching against the invocation history. TOMOYO provides an invocation history-based application-oriented access control for Linux [12]. However, a separate policy is defined for each different invocation string that is allowed, meaning a single program may require many separate policies if it is executed by a number of separate programs [12]. Neither of these systems currently enforce the intersection of policies for separate programs.

4 Evaluation

Substantial effort has been made to evaluate the efficacy of the FBAC model, much of which has previously been published. Here we provide an overview of the results of the evaluation that has been conducted.

4.1 Prototype development

As previously mentioned, a prototype mechanism implementing the FBAC model has been designed and built. The mechanism is known as FBAC-LSM and is implemented for Linux platforms [37]. FBAC-LSM is available as free open source software [38]. The goal of the implementation is to act as a proof of concept and to facilitate evaluation of the FBAC model. As improved usability is a major objective of this research, the user interface aims to leverage the FBAC model constructs to provide policy construction with ease of use. As the FBAC-LSM name implies, a main component is a Linux security module (LSM). As the implementation is intended as a proof of concept, it does not aim to provide complete coverage and has not been verified to be error free.

As described in Section 2, the FBAC model may be implemented as either label-based or name-based. The FBAC-LSM implementation mediates access by means of name-based controls. Using name-based mediation, resources are protected based on their names rather than via labels attached to objects. For example, access to files is mediated in terms of their pathnames rather than the names of labels associated with the files. Examples of other name-based mechanisms are AppArmor [10] and TOMOYO [39], while SELinux [40] and SMACK [41] are examples of label-based LSMs. Some access control model specifications, such as traditional MAC [42] and DTE [43], specify that they are implemented as label-based mechanisms. FBAC allows the model to be implemented either way; the type of mediation used by the FBAC model is not defined by the FBAC specification. Name-based mediation is used by FBAC-LSM because it provides conceptual simplicity, as security is defined in terms of concepts users are familiar with, rather than associations with labels,

which are less familiar to users, and enables access to be granted based on pathname patterns, which can represent complex rules in simple terms.

The prototype has been used to analyse the ability of FBAC to model and enforce the needs of applications, and to study the usability benefits of the scheme. In order to evaluate the practical aspects of the model it was deemed necessary to create an implementation that was grounded in an operating system with existing applications to be confined. As described in the following sections, the results of evaluation have demonstrated that the model can provide improvements for confining existing applications on Linux systems. As the benefits of the implementation can predominantly be attributed to the policy abstraction that the FBAC model provides, we contend that the evaluation results can be attributed to the model, which is platform independent.

4.2 Modelling the Privileges Assigned to Applications

As a preliminary investigation of the suitability of the model, the resource requirements of four different web browsers were analysed, and a hierarchical set of FBAC functionalities were created [44]. Web browsers were chosen as the type of application to study due to their inherent internal complexity and feature richness. The FBAC policy language, FBAC-PL, was developed for the implementation [45], and was able to express the security goals, resulting in a policy that can confine the applications to expected resource use. The resulting policy compared very favourable to other existing schemes, such as SELinux, AppArmor, and Systrace. The Web_Browser functionality provided an abstract way of granting an application the authority to perform web browsing. Parameters were used to adapt the general policy abstraction to the individual requirements of different applications: for example, each application stored had separate configuration directories. Parameters were also used to specify user preferences, such as locations authorised for file downloads. The hierarchical nature of functionalities also enabled encapsulation of policy details, which enabled policy to abstract details away from end user facing configuration, while enabling policy reuse at the functionality development level.

Subsequently, over 100 applications have been profiled and FBAC policies have been created to confine them. Except in a few fringe cases, the functionalities required by applications were obvious and enabled policy reuse across different applications.

4.2.1 Methodology

One hundred and two applications were analysed in terms of resource usage and privilege requirements. Based on this analysis, functionalities were created and expressed using FBAC-PL [45]. These policy abstractions were then used as a basis for constructing policies to restrict the applications to authorised behaviour using FBAC-LSM. Applications were selected for analysis based on the features they provided and their availability on the policy development environment. Games, image editing and viewing programs, video and audio players, text editors, network clients (IRC chat clients, FTP clients, bittorrent clients, and web browsers), and some widely-used command-line programs were analysed.

Developing functionalities, which abstract common privilege requirements in terms of the features the application provides, involved a number of steps. These are listed below and then described in further detail:

– identifying the resources applications utilised;
– identifying the purpose of each of these resources;
– determining whether access to each resource was required for the application to perform the user's intent;
– grouping required resources, based on features provided, into functionalities; and, abstracting away application-specific resources by replacing literal resource descriptions with parameters.

Identifying the resources that applications utilise was performed by executing the applications, exercising their primary features, and using tools to analyse the resources and the type of access requested. Analysis of the resources used by applications was carried out using strace (which outputs the system calls used by programs), AppArmor profiling tools (which output file/type AppArmor rules matching accesses), by analysing the open source application profiles that were available for AppArmor, and using the FBAC-LSM module auditing features and user-space tools. The FBAC-LSM policy manager has a learning mode that interacts with the module. The policy manager was used to interactively add to application policies. The resulting output was then analysed.

Identifying the purpose of each of the accessed resources and determining whether access to each resource was required was an iterative process. Examination of the context of use, source code inspection, examination of resource contents, and web searches were used to identify the reason applications accessed these resources. This information was used to make decisions about whether to authorise access to the resources. When access to a resource seemed unnecessary, the ability of

the application to function without access to the resource was tested. FBAC-PL rules granting access to the required resources were grouped together based on the features provided by applications and the way programs interacted with users. Each of these groups of privileges were either expressed as FBAC-PL high-level functionalities, representing program features, or base-level functionalities, representing types of programs. Application-specific resources were replaced with functionality parameters, thereby abstracting away those details from the functionalities. These application-specific resources were specified for each application policy as arguments to parameters when assigning functionalities. Required privileges that were not appropriate for any functionalities were assigned directly to application policies.

The ability to grant applications direct privileges in addition to those authorised via functionalities is a feature of FBAC-PL which is beyond the FBAC model presented in this paper, and the extent that this was (not) required for policy represents a measure of the extent that the model suited the studied use cases[1].

Privileges within functionalities that described components of high-level features – that is, groups of privileges that were reused in multiple functionalities or with logical relationships – were likewise grouped into low-level functionalities. When appropriate for policy reuse, the parent-functionality-specific details were abstracted using parameters. Thus, low-level functionalities were reused within other functionalities and, when child functionalities had parameters, arguments were specified by the parent-functionality to adapt contained functionalities to the needs of the parent-functionality. The resulting policy structure was a hierarchy of functionalities, starting with the functionalities directly assigned to applications where, for example, a high-level functionality would contain other high-level functionalities and low-level functionalities, which would in turn contain other functionalities, and so on.

4.2.2 Policy Abstraction Results

Using the methodology described, 144 functionalities were created. Functionalities can be assigned "high-level", "base-level", or "low-level" status in the policy language, and this has the benefit of improving the user interface and simplifies discussion. Of the functionalities created, three were base-level (2.1%), 40 were

Table 3 Base-level Functionalities

Base-level Functionalities	Number of Parameters
Simple_Commandline_Program	0
Standard_Commandline_Application	6
Standard_Graphical_Application	7

high-level (27.8%), and 101 were low-level functionalities (70.1%). The functionalities created are available in full online [38]. These results add to the literature describing behavioural classes of programs.

Base-level functionalities were designed to represent the different ways programs interface with users. Based on the programs analysed, the three base-level functionalities and the number of parameters for each of these functionalities is shown in Table 3.

These base-level functionalities contained low-level functionalities. For example, the Standard_Graphical_Application functionality included the functionalities: base, gui, audio, common_console_helper_programs, tmp_access, printer, IPC_system_aware-dbus_system_bus, mime_aware, and other functionalities that grant access to application-specific resources specified as parameters.
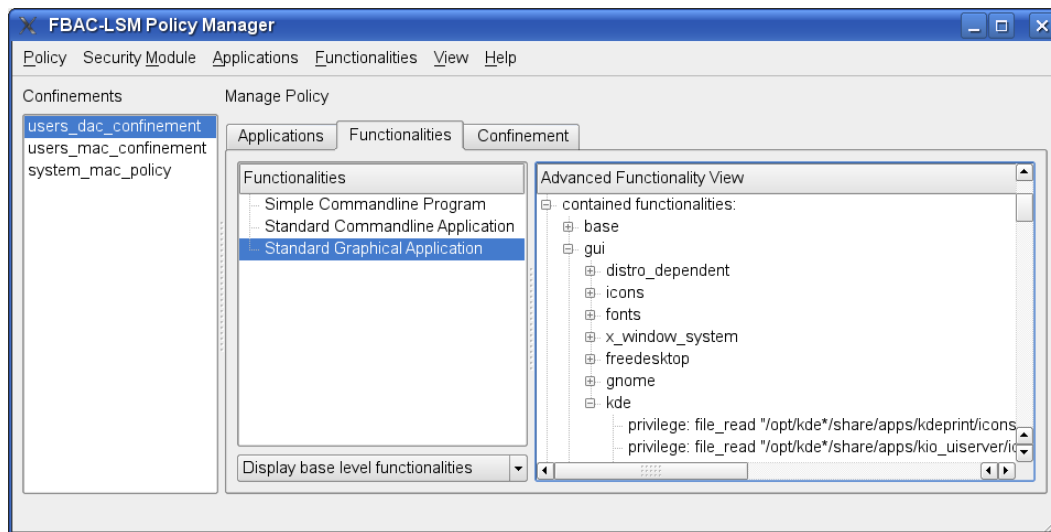
Most of these low-level functionalities were likewise made up of other low-level functionalities. As shown in Figure 13, using the FBAC-LSM policy manager's advanced views one may "drill down" from base or high-level functionalities though the hierarchy of contained functionalities.

The high-level functionalities created were designed to provide the privileges necessary for programs to perform the high-level features identified during analysis. High-level functionalities have been assigned categories to group related functionalities for ease of use. Table 4 shows a list of the 40 high-level functionalities that were developed. The categories into which they were grouped, and the number of parameters for each, are also displayed.

A mean of 1.9 parameters were defined per high-level functionality. This result shows that only a small number of application-specific details are typically required to adapt high-level functionalities to specific applications. FBAC-PL was successfully able to represent feature-based security goals for applications as high-level functionalities.

4.2.3 Policies for Applications

Utilising the functionalities developed, policies were created for the applications. Tables 5 to 12 give an overview of the application policies created. Each table shows the application policy names (which, by convention, were named after the command used to start each applica-

---

[1] Note that the same result can be achieved without deviating from the FBAC model, by creating non-reusable application-specific functionalities. This was avoided during the study, since the study aimed to analyse the reusability of the FBAC policy abstractions.

Fig. 13 The Policy Manager's Advanced Functionality View

tion), the functionalities directly assigned to the application policy, and the number of privileges directly assigned to the application policy. Applications that were assigned the same functionalities with the same number of additional privileges are grouped into the same row.

Of the 102 applications studied, only four required privileges that were deemed unsuited to the functionalities and parameters developed. Two of these four applications (glchess, xchat-gnome) required extra privileges because they acted as front-ends for other applications, one (digikam) stored its configuration files in a non-standard location, and the other (kcheckpass) stored files in an unusual location for a graphical application. As an exception, during policy analysis, the lbreakout game corrupted its configuration files; the result being that even when no confinement policy was in effect, the game crashed with a segmentation fault.

4.2.4 Policy Discussion

The analysis of existing applications and the policy development presented in this paper demonstrate that FBAC functionalities can model and abstract the privileges required to perform program features. The created functionalities were successfully utilised to authorise the applications to perform the features they provided, while severely limiting the damage that may be caused by malware or software vulnerabilities. Results show that the functionality construct is able to provide reusable abstractions that can encapsulate and abstract policy details, can adapt to the needs of specific applications to provide the privileges they require, and reduce the challenges that can arise when applying rule-based application-oriented access controls to confine existing software. Based on these results, we contend that FBAC

provides a practical solution that is an excellent balance between the principle of least privilege and that of psychological acceptability. These results are expected to be operating system independent, should FBAC be re-implemented.

One of the challenges facing all finely-grained rule-based application-oriented access control schemes is policy complexity. Each application policy is typically made up of a large number of low-level access rules, and the complexity of the overall system policy more or less increases in direct relation to the number of applications that are confined. Some existing schemes include simple abstractions that group related rules (for example, AppArmor policy abstractions, and SELinux domains); however, these abstractions only represent relatively low-level aspects of programs and have limited reuse. In contrast, the policies created for FBAC-LSM are largely defined in terms of reusable abstractions. The result is a hierarchical policy configuration that reduces redundancy in both application policies, and also within policy abstractions (where functionalities are defined in terms of other functionalities). The results presented in this paper show that the functionalities developed were reusable and flexible; functionalities were adapted to provide authority to multiple applications and functionalities. FBAC therefore improves the scalability of policy, since confining applications is performed by applying existing functionalities, rather than creating complex low-level rules.

The structure produced by functionality hierarchies provided layers of abstraction. In the policies created, logically grouped rules were successfully abstracted into functionalities. This abstraction improves the manageability of policy, by making it possible to drill down

Table 4 High-level Functionalities

Category	High-level Functionalities	Number of Parameters
File/Media Editor	File_Editor	2
	Archive_Editor	2
	Audio_Editor	2
	Document_Editor	2
	Image_Editor	2
	PDF_Editor	2
	Video_Editor	2
	Web_Files_Editor	2
File Viewer / Media Player	File_Viewer	2
	Archive_Viewer	2
	Audio_Player	2
	Document_Viewer	2
	Image_Viewer	2
	PDF_Viewer	2
	Video_Player	2
	Web_Files_Viewer	2
Game	Game	3
	Network Game	6
Network Client	General Network Client	6
	BitTorrent_Client	4
	Email_Client	6
	Ftp_Client	3
	Downloader	2
	Irc_Chat_Client	4
	News_Reader_Client	1
	ICMP_Pinger	1
	Web_Browser	4
System Tools	Deleter	2
	Process_Information	0
	File_System_Mounter	0
	Uses_Shell	1
	System_Password_Management	0
	System_Password_Check	0
Platform	Uses_Perl	0
	Uses_Mono	0
	Uses_Python	0
	Uses_Orbit	0
	Uses_Java	0
	Uses_Ruby	0
	Uses_XulRunner	0

Table 5 Overview of FTP and Bittorrent Application Policies (file transfer.fbac)

Applications	Functionalities	Additional Privileges
gftp, filezilla	Standard_Graphical_Application Ftp_Client	0
ktorrent, torium, transmission	Standard_Graphical_Application BitTorrent_Client	0
ftp	Standard_Commandline_Application Ftp_Client	0
ncftp, yaftp	Simple_Commandline_Program Ftp_Client	0
wget	Simple_Commandline_Program Downloader	0
deluge	Standard_Graphical_Application BitTorrent_Client Uses_Python	0
rtorrent	Simple_Commandline_Program BitTorrent_Client	0

velopment would yield functionalities that would satisfy these privilege requirements. On the occasions where an application does require access to additional resources (such as access to a particular non-standard resource, or to GPS or other sensors – as may increasingly become popular), the additional access attempts can be vetted separately from the complex finely-grained resource usage that can be authorised based on functionalities that represent the high-level features the application provides.

This research shows that using a functionality-based approach can yield reusable policy abstractions, and that the privilege requirements of applications map well to these abstractions in practice. From a policy development point of view, this builds a strong case for the reusability and flexibility of the FBAC model.

Although providing policy configuration and management benefits, it was evident that at point of end user configuration the specification of parameter arguments was the least intuitive and most complex aspect of the model. Fortunately the model proved to be uniquely suited to policy specification automation.

4.3 Automation and a Priori Policy Specification

Automation techniques have been developed to assist in FBAC policy specification, automating or providing suggestions for each of the steps: selecting executables, associating functionalities, and specifying argu-

from higher level functionalities, to low-level details. This structure can ease maintenance of policy, since details are encapsulated according to the purpose of the rules. As illustrated in Figure 13, this can facilitate visual representations of policy that can assist in policy maintenance. FBAC also separates the task of abstraction development from association with application policies. Assigning functionalities to applications is separated from the more complicated task of developing functionalities, which would normally be performed by someone with more specialised knowledge.

Of the 102 applications studied, only four required privileges in addition to those provided by the functionalities developed. Those that had additional privilege requirements were resolved with minor additions, and it is believed that, in each case, further research and de-

Table 6 Overview of Game Application Policies (games.fbac)

Applications	Functionalities	Additional Privileges
gnobots, gnometris, kasteroids, kfouleggs, kgoldrunner, ksirtet, ksmiletris, ksnake, kspaceduel, ktron, ktumberling, supertux, kenolaba, kbackgammon, kblackbox, gtali, kmahjongg, mahjongg, kreversi, blackjack, kpat, kpoker, sol, ksame, kbounce, konquest, kmines, gnomine, glines, gnotski, samegnome, gnotravex, ksokoban, katomic, kjumpingcube, knetwalk, klines, kolf, brutalchess, xmoto, klickety	Standard_Graphical_-Application Game	0
glchess	Standard_Graphical_-Application Game Uses_Python Uses_Orbit	1
gnome-sudoku	Standard_Graphical_-Application Game Uses_Python Uses_Orbit	0
iagno, gnect, lskat, gnibbles, kwin4, kbattleship	Standard_Graphical_-Application Network_Game	0
frozen-bubble	Standard_Graphical_-Application Network_Game Uses_Perl	0
lbreakout	Standard_Graphical_-Application Game	N/A

Table 7 Overview of Graphics Application Policies (graphics.fbac)

Applications	Functionalities	Additional Privileges
digikam	Standard_Graphical_-Application Image_Editor	2
eog, gimp, krita	Standard_Graphical_-Application Image_Editor	0
karbon	Standard_Graphical_-Application Image_Editor	0 (2+ for parsing other file formats)
gwenview	Standard_Graphical_-Application Image_Viewer	0

Table 8 Overview of IRC Client Application Policies (irc clients.fbac)

Applications	Functionalities	Additional Privileges
konversation, ksirc, dsirc	Standard_Graphical_-Application Irc_Chat_Client Uses_Perl	0
bitchx	Standard_-Commandline_-Application Irc_Chat_Client	0
xchat	Standard_Graphical_-Application Irc_Chat_Client	0
xchat-gnome	Standard_Graphical_-Application Irc_Chat_Client	1

Table 9 Overview of Audio and Video Players Application Policies (media players.fbac)

Applications	Functionalities	Additional Privileges
amarok	Standard_Graphical_-Application Audio_Player Uses_Ruby	0
banshee	Standard_Graphical_-Application Audio_Player Video_Player Uses_Mono	0
codeine, gmplayer, kaffeine, kplayer, realplay, vlc, xine, mplayer	Standard_Graphical_-Application Audio_Player Video_Player	0
totem	Standard_Graphical_-Application Video_Player	0

Table 10 Overview of File Editor Application Policies (text editors.fbac)

Applications	Functionalities	Additional Privileges
gedit, kate, kwrite	Standard_Graphical_-Application File_Editor	0
vi	Standard_-Commandline_-Application	0

ments for each functionality parameter [46]. Automation is achieved via simple analysis of program dependencies, program management information, and filesystem contents. For this purpose, FBAC-PL includes automation metadata within the definition of functionalities and parameters [45]. For example, functionality definitions can contain libraries and desktop icon categories that indicate the functionality is likely to be appropriate. As a result of automation, complete (and

Table 11 Overview of Web Brower Application Policies (web browsers.fbac)

Applications	Functionalities	Additional Privileges
epiphany	Standard_Graphical_-Application Web_Browser	0
firefox	Standard_Graphical_-Application Web_Browser Uses_XulRunner	0
lynx	Standard_-Commandline_-Application Web_Browser	0
opera	Standard_Graphical_-Application Web_Browser Email_Client Irc_Chat_Client BitTorrent_Client News_Reader_Client	0

Table 12 Overview of Common Console Application Policies (console.fbac)

Applications	Functionalities	Additional Privileges
rm	Simple_-Commandline_-Program Deleter	0
cat, ls	Simple_-Commandline_-Program File_Viewer	0
ps	Simple_-Commandline_-Program Process_Information	0
mount	Simple_-Commandline_-Program File_System_Mounter	0
passwd, expiry, gpasswd	Simple_-Commandline_-Program System_Passwd_-Management	0
kcheckpass	Standard_Graphical_-Application System_Passwd_-Check	3

near complete) policies can easily be created a priori: that is, without first executing the program being confined. During assessment, policy only occasionally suffered from rare false negatives or positives.

Due to the complexity of rule-based application-oriented access controls, learning modes are often used to generate policy. Schemes that typically rely on learning modes to generate policy include Systrace [9], Ap-pArmor [10], SELinux [11], and TOMOYO [12]. Learning modes generate policy rules by recording application behaviour. Disadvantages of this approach include the fact that this recording is typically done while the application is not confined, meaning the typical approach is not appropriate for confining potentially malicious software. Conversely, if an application is confined during development, typically many iterations of policy development will be required, since many programs continue to crash until policy is somewhat complete. Furthermore, any policy specification exposes the user to low level privilege requirements of applications and many users are not likely to possess the knowledge necessary. In contrast, automation that leverages FBAC model features is able to generate policies that are relatively easy to comprehend, and enable the user to enforce their security goals in relation to the application without exposing them to low level policy details. During evaluation, the few false negatives that prevented legitimate behaviour could be resolved using FBAC-LSM learning mode, which involves fewer decisions than other schemes as the learning mode is only required for these fringe cases [46].

These automation techniques successfully further lowered the expertise required to construct policies to confine applications using FBAC-LSM, and added further evidence of the benefits of the FBAC model.

4.4 Usability

A comparative study was conducted to evaluate the usability of the FBAC model, by comparing FBAC-LSM with two widely deployed alternative systems that provide application-oriented controls, AppArmor, and SELinux [47,48]. After a pilot study, 39 participants completed the experiment, using all three security mechanisms to construct policies to confine two programs. Descriptive and inferential statistics were utilised to compare the within-subjects effects of the three security systems.

As previously reported [48][2], a one-way within subjects ANOVA was conducted to compare the effect of security system on System Usability Scale (SUS) scores. The security system was found to have a significant effect, Wilks' Lambda = 0.38, $F_{(2,35)}$ = 28.99, p < .001, n=37. Post hoc analysis using the Tukey LSD test showed that the usability of all three systems were significantly different from each other. On average FBAC-LSM received the highest SUS scores (M=70.21), fol-

---

[2] For further details regarding the methods, results, and discussion please refer to this publication. An overview is presented here as part of the wider evaluation of the FBAC model.

lowed by AppArmor (M=54.93), and SELinux (M=34.58). On average, participants also ranked FBAC-LSM easiest to use, easiest to understand, and most likely to use again. These results indicated that FBAC-LSM demonstrated higher perceived usability compared to AppArmor and SELinux.

The effect of the security system on the participants success in creating policies that were actually in effect was analysed using repeated measures logistic regression. In each case there was a significant effect, $p < .007$, and in each case FBAC-LSM was most frequently successfully used to create enforcing policies. In the web browser example, 90% of participants successfully created a policy that was enforced using using FBAC-LSM, 66% using AppArmor, and only 23% using SELinux. In the Trojan horse simulation example 82% of participants created enforced policies using FBAC-LSM, 71% using AppArmor, and only 22% using SELinux. The results showed that, compared to the other systems, FBAC-LSM had significantly higher success rates for both policy creation and enforcement. It was also found that using FBAC-LSM more of the policies that were created allowed programs to function correctly while they were confined.

The final security state of the systems were further analysed using the non-parametric Friedman test (since not all assumptions for ANOVA were met), and the security system was found to have a significant effect ($p < 0.001$) on the overall risk exposure. Risk exposure was measured in terms of the number of specific security sensitive resources that the participant-configured security systems did not prevent access to. Post hoc analysis showed that the policies created using FBAC-LSM had a lower risk exposure (M=14.3) compared to both AppArmor (M=30.3) and SELinux (M=43.0). The study also demonstrated that FBAC-LSM provided similar security benefits when successfully confining benign software, such as a web browser. However, using FBAC-LSM, participants more frequently successfully created policies, thereby reducing risk in the case of confining trustworthy software to protect against vulnerabilities. The policies created using FBAC-LSM provided significantly increased protection against malware, compared to the other mechanisms.

Participants also rated FBAC-LSM as being more time efficient; however, the time recorded for AppArmor profiling was often shorter. This discrepancy was attributed to the behaviour of participants who rapidly clicked through the many AppArmor dialogues, which in the case of the Trojan horse simulation also had the effect of inadvertently authorising malicious behaviour.

As mentioned, the usability study showed empirically that FBAC-LSM was rated as significantly more usable than the other systems. FBAC-LSM was also significantly more successful at securing the systems. Learning mode systems, which rely on users vetting rules, were demonstrated to be unreliable and open to subversion by malicious programs compared to a functionality-based approach.

Qualitative analysis provided further insight into factors that can influence the usability of application-oriented access controls, and confirmed that the abstractions and techniques enabled by the FBAC model contributed to usability benefits [47].

5 Conclusion

This paper has proposed and defined a novel rule-based application-oriented access control model, FBAC, that confines processes based on application policies constructed using reusable and adaptable policy abstractions known as functionalities. Processes can simultaneously be subject to restrictions, known as confinements, that can enforce mandatory and discretionary application policies specified by separate users.

As discussed, the FBAC model has many features unique within the field of application-oriented access controls, including:

- hierarchical policy primitives;
- parameterised policy abstractions that can be combined and layered;
- simultaneous enforcement of multiple security goals/ sets of policies, which can enforce a diverse range of types of application restrictions;
- dynamic activation and deactivation of logically grouped portions of a processes authority;
- process invocation history intersection-based privilege propagation.

FBAC was designed to provide application confinement that is functionality-based in nature, capable of modelling high level security goals for application restrictions, with abstractions that encapsulate low level policy details. This design separates the task of specifying application-functionality associations from the more involved task of specifying security rules for classes of programs.

The Linux prototype, FBAC-LSM, has enabled the evaluation of the efficacy of the FBAC model, and demonstrates the usability advantages and unique features of this new approach to application confinement. Analysis showed that the FBAC model is capable of representing the privilege needs of applications. The model is also well suited to automation techniques that can in many cases create complete application policies without first running the applications. This is an improvement

over previous approaches that typically rely on learning modes to generate policies. A usability study was conducted, which showed that compared to two widely-deployed alternatives (SELinux and AppArmor), FBAC-LSM had significantly higher perceived usability and resulted in significantly more protective policies.

It is our hope that the model presented in this paper will be used to protect end users against malicious code, and will help to direct and inspire future work in this developing field of research.

References

1. Yee, B., Sehr, D., Dardyk, G., Chen, J.B., Muth, R., Ormandy, T., Okasaka, S., Narula, N., Fullagar, N.: Native Client: A Sandbox for Portable, Untrusted x86 Native Code. Communications of the ACM 53(1), 91-99 (2010)
2. Gong, L., Mueller, M., Prafullchandra, H., Schemers, R.: 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 1997. Prentice Hall PTR
3. Whitaker, A., Shaw, M., Gribble, S.D.: Denali: Lightweight Virtual Machines for Distributed and Networked Applications. In: 5th USENIX Symposium on Operating Systems Design and Implementation, Boston, MA, USA 2002. USENIX Association
4. Madnick, S.E., Donovan, J.J.: Application and Analysis of the Virtual Machine Approach to Information Security. In: ACM Workshop on Virtual Computer Systems, Cambridge, MA, USA 1973. Harvard University
5. Kamp, P.-H., Watson, R.: Jails: Confining the Omnipotent Root. In: 2nd International System Administration and Networking Conference (SANE 2000), Maastricht, The Netherlands 2000
6. Tucker, A., Comay, D.: 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. Boebert, W.E., Kain, R.Y.: A Practical Alternative to Hierarchical Integrity Policies. In: 8th National Computer Security Conference, Gaithersburg, MD, USA 1985. NIST
8. Goldberg, I., Wagner, D., Thomas, R., Brewer, E.A.: A Secure Environment for Untrusted Helper Applications: Confining the Wily Hacker. In: 6th USENIX Security Symposium, San Jose, CA, USA 1996. USENIX Association
9. Provos, N.: Improving Host Security with System Call Policies. In: 12th USENIX Security Symposium, Washington, DC, USA, August 2002. USENIX Association
10. Cowan, C., Beattie, S., Kroah-Hartman, G., Pu, C., Wagle, P., Gligor, V.: SubDomain: Parsimonious Server Security. In: USENIX 14th Systems Administration Conference, New Orleans, LA, USA 2000. USENIX Association
11. Loscocco, P., Smalley, S.: Integrating Flexible Support for Security Policies into the Linux Operating System. In: FREENIX Track: 2001 USENIX Annual Technical Conference, Boston, MA, USA 2001. USENIX Association
12. Harada, T., Horie, T., Tanaka, K.: Task Oriented Management Obviates Your Onus on Linux. In: Linux Conference 2004, Tokyo, Japan 2004
13. Sandhu, R., Ferraiolo, D., Kuhn, R.: Role Based Access Control. In. American National Standards Institute / International Committee for Information Technology Standards (ANSI/INCITS), (2004)
14. Walker, K., Sterne, D., Badger, M., Petkac, M., Sherman, D., Oostendorp, K.: Confining Root Programs with Domain and Type Enforcement. In: 6th USENIX Security Symposium, San Jose, CA, USA 1996. USENIX Association
15. Schreuders, Z.C.: Thesis. A Role-Based Approach to Restricting Application Execution. Murdoch University (2005)
16. Raje, M.: TRCS 99-12: Behavior-based Confinement of Untrusted Applications. University of Calfornia (1999)
17. Acharya, A., Raje, M.: MAPbox: Using Parameterized Behavior Classes to Confine Applications. In: 9th USENIX Security Symposium, Denver, CO, USA 2000. USENIX Association
18. Giuri, L., Iglio, P.: Role Templates for Content-based Access Control. In: 2nd ACM Workshop on Role-based Access Control, Fairfax, VA, USA 1997. ACM Press
19. Yao, W., Moody, K., Bacon, J.: A Model of OASIS Role-based Access Control and its Support for Active Security. In: 6th ACM Symposium on Access Control Models and Technologies, Chantilly, VA, USA 2001. ACM Press
20. Ferraiolo, D., Cugini, J.A., Kuhn, R.: Role-Based Access Control (RBAC): Features and Motivations. In: 11th Annual Computer Security Applications Conference (ACSAC), Gaithersburg, MD, USA 1995. IEEE Computer Society Press
21. Johnson, M., Karat, J., Karat, C.-M., Grueneberg, K.: Optimizing a Policy Authoring Framework for Security and Privacy Policies. In: 6th Symposium on Usable Privacy and Security (SOUPS), Redmond, Washington, DC, USA 2010. ACM Press
22. Wagner, D.A.: Janus: An Approach for Confinement of Untrusted Applications. In: Electrical Engineering and Computer Sciences. University of California, Berkeley, CA, USA, (1999)
23. Berman, A., Bourassa, V., Selberg, E.: TRON: Process-Specific File Protection for the UNIX Operating System. In: Winter USENIX Conference, New Orleans, LA, USA 1995. USENIX Association
24. Hallyn, S.E., Kearns, P.: Domain and Type Enforcement for Linux. In: 4th Annual Linux Showcase and Conference, Atlanta, GA, USA 2000
25. Zanin, G., Mancini, L.V.: Towards a Formal Model for Security Policies Specification and Validation in the SELinux System. In: 9th ACM Symposium on Access Control Models and Technologies, Yorktown Heights, NY, USA 2004. ACM Press
26. Hallyn, S.E., Morgan, A.G.: Linux Capabilities: Making Them Work. In: The Linux Symposium, Ottawa, ON, Canada 2008
27. Edge, C., Barker, W., Hunter, B., Sullivan, G.: Enterprise Mac Security: Mac OS X Snow Leopard, Second Edition. Apress, (2010)
28. Ferraiolo, D.F., Sandhu, R., Gavrila, S., Kuhn, D.R., Chandramouli, R.: Proposed NIST Standard for Role-Based Access Control. ACM Transactions on Information and System Security (TISSEC) 4(3), 224-274 (2001)
29. Tidswell, J., Potter, J.: An Approach to Dynamic Domain and Type Enforcement. In: Australasian Conference on Information Security and Privacy, Syndey, NSW, Australia 1997. Springer

30. Ott, A.: The Role Compatibility Security Model. In: 7th Nordic Workshop on Secure IT Systems (NordSec), Karlstad, Sweden 2002
31. Hinrichs, S., Naldurg, P.: Attack-based Domain Transition Analysis. In: 2nd Annual Security Enhanced Linux Symposium, Baltimore, MD, USA 2006
32. Hardy, N.: The Confused Deputy: Or Why Capabilities Might Have Been Invented. ACM SIGOPS Operating Systems Review 22(4), 36-38 (1988)
33. Fournet, C., Gordon, A.D.: Stack Inspection: Theory and Variants. ACM Transactions on Programming Languages and Systems (TOPLAS) 25(3), 360-399 (2003)
34. Wallach, D.S., Felten, E.W.: Understanding Java Stack Inspection. In: 19th IEEE Symposium on Security and Privacy, Oakland, CA, USA 1998. IEEE Computer Society
35. Besson, F., Blanc, T., Fournet, C., Gordon, A.D.: From Stack Inspection to Access Control: A Security Analysis for Libraries. In: 17th IEEE Computer Security Foundations Workshop, Asilomar, CA, USA 2004. IEEE Computer Society
36. Hunt, G., Larus, J., Abadi, M., Aiken, M., Barham, P., Fhndrich, M., Hawblitzel, C., Hodson, O., Levi, S., Murphy, N., Steensgaard, B., Tarditi, D., Wobber, T., Zill, B.: An Overview of the Singularity Project. In. Microsoft Research, Redmond, WA, USA, (2005)
37. Schreuders, Z.C.: The Functionality-Based Application Confinement Model and its Linux Prototype FBAC-LSM (Presentation). In: linux.conf.au - LCA2009, Tasmania, Australia 2009
38. Schreuders, Z.C.: FBAC-LSM: Protect Yourself From Your Apps. http://schreuders.org/FBAC-LSM (Accessed 2011)
39. Harada, T., Horie, T., Tanaka, K.: Towards a Manageable Linux Security. In: Linux Conference 2005 (Japanese), Japan 2005
40. Morris, J.: Filesystem Labeling in SELinux. Linux Journal(126), 22-24 (2004)
41. Schaufler, C.: The Simplified Mandatory Access Control Kernel. In: http://schaufler-ca.com/. (2008)
42. Department of Defense: Trusted Computer Security Evaluation Criteria. DOD 5200.28-STD. (1985)
43. Boebert, W.E., Kain, R.Y.: A Practical Alternative to Hierarchical Integrity Policies. Proceedings of the 8th National Computer Security Conference, 18-27 (1985)
44. Schreuders, Z.C., Payne, C.: Reusability of Functionality-Based Application Confinement Policy Abstractions. In: 10th International Conference on Information and Communications Security (ICICS 2008), Birmingham, UK 2008. Springer
45. Schreuders, Z.C., Payne, C., McGill, T.: A Policy Language for Abstraction and Automation in Application-oriented Access Controls: The Functionality-based Application Confinement Policy Language. In: IEEE International Symposium on Policies for Distributed Systems and Networks (POLICY 2011), Italy, Pisa 2011. IEEE Computer Society
46. Schreuders, Z.C., Payne, C., McGill, T.: Techniques for Automating Policy Specification for Application-oriented Access Controls. In: 6th International Conference on Availability, Reliability and Security (ARES 2011) Vienna, Austria 2011. IEEE Computer Society
47. Schreuders, Z.C., McGill, T., Payne, C.: Towards Usable Application-oriented Access Controls: Qualitative Results from a Usability Study of SELinux, AppArmor and FBAC-LSM. International Journal of Information Security and Privacy 6(1), 57-76 (2012)
48. Schreuders, Z.C., McGill, T., Payne, C.: Empowering End Users to Confine Their Own Applications: The Results of a Usability Study Comparing SELinux, AppArmor and FBAC-LSM. ACM Transactions on Information and System Security (TISSEC) 14(2), 1-28 (2011)

Appendix: Model Logic

This paper has introduced FBAC, defined its components and the model structure, discussed the design of privilege propagation across process invocation, and presented evaluation of the model. This appendix defines the logic that is utilised to enforce the FBAC model. Algorithms are presented here for starting new processes, performing access decisions, and resolving parameters to values. The specification of these algorithms is necessarily complex; however, each algorithm is also described in less formal terms.

The names of the functions used to identify mapped entities in the following algorithms are displayed on the entity mapping connections in Figure 14. For example, PT(PROCESS x) represents the set of task confinements associated with the process denoted by the variable 'x'. This notation can denote the association in either direction: for example, PT(TASK_CONFINEMENT c) represents the process associated with the task confinement 'c'. Attribute values (as shown in Figure 14 as dotted rectangles) are accessed using a "." followed by the entity name: for example, task_confinement.propagate. It is suggested the reader refer to the figure while reading the following sections.

A.1 Starting a Process

The procedure for starting a new process involves checking that the program is authorised to run, creating the process context and policy instances, and managing the ancestry propagation of authority. The procedure is divided into functions, each function is first described then defined using pseudocode.

As defined in Figure 15, the function process_start describes how processes start. The function parent_task_conf is called to retrieve each task confinement from the parent, which is added to the ancestry association for the child process. The function find_propagation_type is then used to retrieve the execute operation that authorises the parent to start the new process. If, for any of the task confinements, the parent is not authorised to start the new process, the attempt is denied and the process will not start. If the process has no parent (that is, the parent is unconfined or nonexistent), then the child starts using the execute_load_profile operation. This means that ancestry for that process will not be considered again when making access decisions.

The application that applies to the process is identified based on the executable path using the find_application function. If no policy is found for the process, the confinement attribute no_profile (which can be configured by an administrator, and was described

```
Function parent_task_conf
Parameters: child_conf, parent
Returns: TASK_CONFINEMENT

TASK_CONFINEMENT retval = NO_PARENT_CONF
for each parent_task_conf in set PT(PROCESS parent)
     if(parent_task_conf inherited_from(child_conf))
          retval = parent_task_conf
return retval
```

Fig. 16 Function parent_task_conf

in Section 2.4) is used to decide whether to restrict the process to a limited policy, confine it using its parent's policy (unconfined if the parent was unconfined), or deny the execution. If the program is permitted to run, the function manage_propagation is then called to modify the process ancestry (if required due to a parent running with the execute_shell operation).

The remaining functions described in this section perform other steps necessary for process_start to function. The process ancestry is maintained separately for each task confinement, since each confinement can represent separate rules. Therefore, a process can have a different ancestry of rules that needs to be considered for each task confinement. The function parent_task_conf, shown in Figure 16, simply retrieves and returns the parent task confinement for a given task confinement, by searching for one that is associated with the parent process and is inherited from the same confinement.

The function find_propagation_type, shown in Figure 17, queries a task confinement for the authority to start a program. If no privilege allows the program to run, the function returns NONE and process_start stops the process from starting. Otherwise, the operation used to authorise the process to start is returned. As shown in the algorithm below, the operations are checked in the following order of precedence: execute_as_current_app, execute_as_interpreted, execute_shell, execute_load_profile, then execute. The function conf_has_authority (as defined later in the access control decision logic) is used to check if starting the program with one of these operations is authorised. If the parent was started using execute_shell or execute_as_current_app, then the confinement's parent's privileges are queried (i.e. the parent's parent, and so on recursively).

The find_application function, shown in Figure 18, simply searches all the application in a confinement, for an application that has an executable matching the executable_path argument.

The function build_task_tree, shown in Figure 19, creates the new records and establishes the relationships between records to represent the presence of a new process on the system. As shown in the algorithm, the new process is first associated with the user entity.
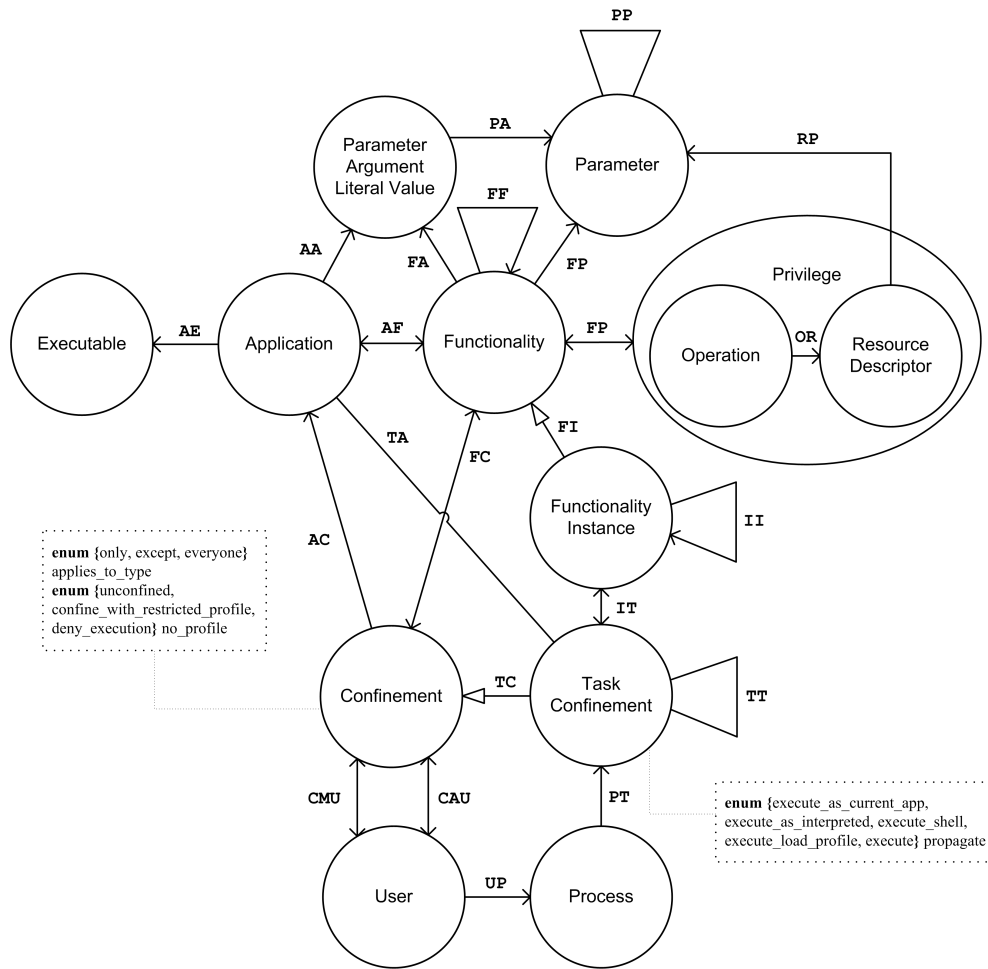
Fig. 14 The FBAC Model, Showing Set Association Function Names

Function process_start
Parameters: path, user, parent, newprocess
Returns: Boolean

```
boolean permit = TRUE
for each conf in set CAU(USER user)
    parent_conf = parent_task_conf(conf, parent)
    if(parent_conf != NO_PARENT_CONF)
        add parent_conf to set TT(CHILD_TASK_CONFINEMENT conf)
        op = find_propagation_type(path, parent_conf)
    else
        op = execute_load_profile
    if(op != NONE)
        app = find_application(path, conf)
        if(app == NOT_FOUND)
            switch(conf.no_profile)
                case confine_with_restricted_profile:
                    permit = build_task_tree (conf, RESTRICTED_APPLICATION, newprocess, user, execute)
                case unconfined:
                    op = execute_as_current_app
                    permit = TRUE
                case deny_execution:
                    permit = FALSE
            else
                permit = build_task_tree(conf, app, newprocess, user, op)
        manage_propagation(op, conf, parent_conf)
    else
        permit = FALSE
```

Fig. 15 Function process_start

```
Function find_propagation_type
Parameters: executable_path, confinement
Returns: OPERATION {execute_as_current_app, execute_as_interpreted, execute_shell, execute_load_profile, or execute}

prop_type = NONE
set execute_operations {execute_as_current_app, execute_as_interpreted, execute_shell, execute_load_profile, execute}
if(confinement.propagate == execute_as_current_app OR confinement.propagate == execute_shell)
      prop_type = find_propagation_type(TT (CHILD_TASK_CONFINEMENT confinement), path)
else
      for each eop in execute_operations
            if(conf_has_authority(eop, executable_path, conf) == TRUE)
                  prop_type = eop
                  break for
return prop_type
```

Fig. 17 Function find_propagation_type

```
Function find_application
Parameters: executable_path, confinement
Returns: APPLICATION

for each app in set AC(CONFINEMENT confinement)
      for each exec in set AE(APPLICATION app)
            if(exec matches(executable_path))
                  return app
return NOT_FOUND
```

Fig. 18 Function find_application

A new task confinement is created as an instance of the confinement and is associated with the process. The application is associated with the new task confinement. All the functionalities associated with the application (and recursively all contained functionalities) are created as functionality instances and associated with the new process.

The function create_func_instance_recursive, shown in Figure 20, is called by build_task_tree, and simply creates and establishes the relationships between contained functionality instances.

After everything else has successfully been established, the function manage_propagation, shown in Figure 21, is called by process_start to set the propagation type of the new process. If the parent confinement was running as execute_shell and the operation used to authorise the process to start was execute_load_profile, then the operation is reset to execute (so that the shell can be used to launch helper programs, as described in Section 3). The operation that has been calculated is associated with the task confinement.

A.2 Access Decision

The logic used to make access control decisions is defined in this section. The function task_has_privilege, shown in Figure 22, is the interface to the decision logic. It reports whether a process is authorised to perform an action, given the operation and the specifics of the resources to be accessed (specified via the 'arguments' function parameter). As shown in the algorithm, task_has_privilege uses the conf_has_authority function to check that every task confinement for a process authorises the proposed access. This enforces the requirement that the resulting permission is an intersection of these confinements.

The function conf_has_authority, shown in Figure 23, returns whether a given task confinement authorises the action. This function takes the type of propagation into account. Depending on the type of propagation, it may call itself recursively for its parent and may check the privileges associated with the task confinement using the test_all_app_privileges_with_op function. If there is no task confinement supplied (that is, there is no parent task confinement), the recursive call returns TRUE, and the intersection of the previous task confinements is returned. This enforces the requirement that the authority granted by a task confinement is an intersection of the hierarchy of task confinements for the process's ancestry.

The function test_all_app_privileges_with_op, shown in Figure 24, is used by conf_has_authority to check whether an application for a specific task confinement grants the authority to perform an action. This is achieved by calling the function test_all_func_privileges_with_op to check whether any of the functionality instances associated with a task confinement's application authorise the activity. Subsequently test_all_func_privileges_with_op, shown in Figure 25, recursively calls itself to check if any functionality instances contained within the functionality grant the access. For every functionality, test_direct_privs is used to test the privileges directly assigned to each functionality.

The function test_direct_privs, shown in Figure 26, searches for privileges matching the specified operation,

Function build_task_tree
Parameters: confinement, application, newprocess, parent, user, op

add newprocess to set UP(USER user)
create task_confinement(confinement)
add task_confinement to set PT(PROCESS newprocess)
add application to set TA(TASK_CONFINEMENT task_confinement)

for each func in set AF(APPLICATION application)
    create_func_instance_recursive(func, task_confinement)

Fig. 19 Function build_task_tree

Function create_func_instance_recursive
Parameters: functionality, task_confinement

create func_instance(functionality)
add func_instance to set IT((TASK_CONFINEMENT task_confinement)
for each func in set II(PARENT_FUNCTIONALITY functionality)
    create_func_instance_recursive(func, task_confinement)

Fig. 20 Function create_func_instance_recursive

Function manage_propagation
Parameters: op, child_conf, parent_conf

if(parent_conf != NO_PARENT_CONF AND parent_conf.propagate == execute_shell AND op == execute_load_profile)
    op = execute
    child_conf.propagate = op

Fig. 21 Function manage_propagation

Function task_has_privilege
Parameters: operation, arguments, current_process
Returns: Boolean

boolean permit = TRUE
for each task_confinement in set PT(PROCESS current_process)
    permit = permit AND conf_has_authority(operation, arguments, process, task_confinement)
    return permit

Fig. 22 Function task_has_privilege

Function conf_has_authority
Parameters: operation, arguments, task_confinement
Returns: Boolean

boolean permit
if(task_confinement == $\emptyset$)
    return TRUE
switch(task_confinement.propagate)
    case execute:
        permit = conf_has_authority(operation, arguments, TT(CHILD_TASK_CONFINEMENT task_confinement) AND test_all_app_-
privileges_with_op(operation, TA(APPLICATION task_confinement), arguments)
    case execute_as_interpreted:
        permit = conf_has_authority(operation, arguments, TT(CHILD_TASK_CONFINEMENT task_confinement) OR test_all_app_-
privileges_with_op(operation, TA(APPLICATION task_confinement), arguments)
    case execute_shell:
        permit = conf_has_authority(operation, arguments, TT(CHILD_TASK_CONFINEMENT task_confinement)
    case execute_as_current_app:
        permit = conf_has_authority(operation, arguments, TT(CHILD_TASK_CONFINEMENT task_confinement)
    case execute_load_profile:
        if(TT(CHILD_TASK_CONFINEMENT task_confinement).propagate == execute_shell)
            permit = conf_has_authority(operation, arguments, TT(CHILD_TASK_CONFINEMENT task_confinement) AND test_all_app_-
privileges_with_op(operation, TA(APPLICATION task_confinement), arguments)
        else
            permit = test_all_app_privileges_with_op(operation, task_confinement, arguments)
return permit

Fig. 23 Function conf_has_authority

```
Function test_all_app_privileges_with_op
Parameters: operation, task_confinement, arguments
Returns: Boolean

boolean permit = FALSE
for each functionality in set IT(TASK_CONFINEMENT task_confinement) while permit == FALSE
    permit = test_all_func_privileges_with_op(operation, functionality, arguments)
    if(permit == TRUE)
        break for
    return permit
```

Fig. 24 Function test_all_app_privileges_with_op

```
Function test_all_func_privileges_with_op
Parameters: operation, functionality, arguments
Returns: Boolean

Boolean permit
permit = test_direct_privs(operation, FP(FUNCTIONALITY FI(FUN_INSTANCE functionality)), arguments)
if(!permit)
    for each func_child in set II(FUNCTIONALITY functionality)
        permit = test_all_func_privileges_with_op(operation, func_child, arguments)
        if(permit == TRUE)
            break for
return permit
```

Fig. 25 Function test_all_func_privileges_with_op

```
Function test_direct_privs
Parameters: operation, privileges, arguments
Returns: Boolean

Boolean permit = FALSE
for each privilege in set privileges
    if(privilege.operation == operation)
        if(test_permission(operation, privilege, arguments) == TRUE)
            permit = TRUE
            break
return permit
```

Fig. 26 Function test_direct_privs

and calls test_permission to test if that privilege grants access.

The function test_permission, shown in Figure 27, calls the implementation dependent has_privilege to check if the arguments supplied to the decision logic are a match for any resource descriptors associated with the privilege. This can involve different types of comparisons based on the type of resource being accessed. As previously discussed in Section 2, this may involve pattern matching between the patterns specified in resource descriptors and the strings representing the resource in the parameter arguments. Resolving parameters, as described in the next section, can be used at this stage to determine non-literal parameter argument values.

A.3 Resolving Parameter Arguments to Privileges

As previously discussed in Section 2.3, application policies or functionalities can supply literal arguments (in other words, actual descriptions of resources) to parameters. These associations are represented in Figure 14 as AA and FA. However, parameters can also refer to other parameters contained in parent functionalities. This is represented as PP in the figure. It is necessary to resolve these parameters to literal arguments in order to make access decisions.

The algorithm presented in this section resolves resource descriptors that refer to parameters to the literal argument values that describe the resources the operation grants access to.

The function resolve_nonliteral_resource_descriptor, shown in Figure 28, returns a literal result that describes resources. If the 'resource_descr' argument is already literal (that is, it is not associated with a parameter), it is simply returned. If it is non-literal (it is associated with a parameter) then the function resolve_argument is used to return the literal parameter argument.

The function resolve_argument, shown in Figure 29, checks whether a parameter is associated with a parent parameter. If it is, the function calls itself recursively until all parents have been traversed and a literal value is found. If the parameter has no parent parameters, then the corresponding literal argument is returned.

Function test_permission
Parameters: operation, privilege, arguments
Returns: Boolean

return [has_privilege](operation, privilege, arguments)

Fig. 27 Function test_permission

Function resolve_nonliteral_resource_descriptor
Parameters: resource_descr
Returns: ARGUMENT or RESOURCE_DESCRIPTOR (if literal)

param = OR(RESOURCE_DESCRIPTOR resource_descr)
if param != ∅
        return resolve_argument(param)
else
        return resource_descr

Fig. 28 Function resolve_nonliteral_resource_descriptor

Function resolve_argument
Parameters: parameter
Returns: ARGUMENT

parent_param = PP(CHILD_PARAMETER parameter)
if parent_param != ∅
        return PA(PARAMETER parameter)
else
        return resolve_argument(parent_param)

Fig. 29 Function resolve_argument