Verification of Resilience Policies that Assist Attribute Based Access Control

Antonios Gouglidis
School of Computing and Communications
Lancaster, UK
a.gouglidis@lancaster.ac.uk

Vincent C. Hu
Computer Security Division
NIST, USA
vincent.hu@nist.gov

Jeremy S. Busby
Department of Management Science
Lancaster, UK
j.s.busby@lancaster.ac.uk

David Hutchison
School of Computing and Communications
Lancaster, UK
d.hutchison@lancaster.ac.uk

ABSTRACT

Access control offers mechanisms to control and limit the actions or operations that are performed by a user on a set of resources in a system. Many access control models exist that are able to support this basic requirement. One of the properties examined in the context of these models is their ability to successfully restrict access to resources. Nevertheless, considering only restriction of access may not be enough in some environments, as in critical infrastructures. The protection of systems in this type of environment requires a new line of enquiry. It is essential to ensure that appropriate access is always possible, even when users and resources are subjected to challenges of various sorts. Resilience in access control is conceived as the ability of a system not to restrict but rather to ensure access to resources. In order to demonstrate the application of resilience in access control, we formally define an attribute based access control model (ABAC) based on guidelines provided by the National Institute of Standards and Technology (NIST). We examine how ABAC-based resilience policies can be specified in temporal logic and how these can be formally verified. The verification of resilience is done using an automated model checking technique, which eventually may lead to reducing the overall complexity required for the verification of resilience policies and serve as a valuable tool for administrators.

CCS Concepts

•General and reference → Verification; •Security and privacy → Formal security models; Access control; •Software and its engineering → Model checking;

ABAC'17, March 24 2017, Scottsdale, AZ, USA
© 2017 ACM. ISBN 978-1-4503-4910-9/17/03...$15.00
DOI: http://dx.doi.org/10.1145/3041048.3041049

Keywords

Attribute based access control; resilience

1. INTRODUCTION

Access control is an essential technique in all computing systems. Its main role is to control and limit the actions or operations in a system that are performed by a user on a set of resources. Access control policies, models and mechanisms are considered to be three abstractions of control introduced by access control systems [10]. These levels of abstraction are responsible for the enforcement of access control policies, as well as for preventing the access policy from subversion. Specifically, a policy can be defined as a high-level requirement that specifies how a user may access a specific resource and how. Access control policies can be enforced in a system through an access control mechanism. The latter is responsible for permitting or denying a user access to a resource, and specifying the nature of access that is permitted. An access control model can be defined as an abstract container of a collection of access control mechanism implementations. These are capable of preserving support for the reasoning of the system policies through a conceptual framework. Hence, the abstraction gap between the mechanism and policy in a system is bridged by means of access control model [24].

The importance of access control in systems led to research in several directions, one of them being the investigation of properties related to the security offered by policies, e.g. secure inter-operation [11, 12, 26]. However, little attention has been paid to the verification of resilience specifications in access control policies. Resilience in access control is conceived as the ability of a system not to restrict, but to enable access to resources [21]. Most of the research work in this context is initiated around the *'resiliency checking problem'*, which examines whether a given resilience policy is satisfied by an access control state. This problem has been investigated from a generic point of view [21], and thus the proposed approaches are agnostic to the actual type of policies implemented by an underlying model. Additional research on the resiliency checking problem was performed to investigate the time complexity introduced by the various parameters used in it [8]. Moreover, the *'resiliency checking* *problem'* is shown to have a connection with the *'work-flow satisfiability problem'* in [9], with the latter being investigated extensively in the literature, e.g. in [6, 7, 27] amongst others. Furthermore, information on work-flow management systems and on how to model and enforce resilience policies is available in [2, 4].

Despite of several research directions being followed to address the problem of ensuring resilience in access control policies, they assume the construction of a resilience aware policy during the design phase of a system, and not during its operational phase. This does not negate the correctness of these approaches, but questions the level of usability provided by them, as well as their applicability in real operational environments. The need for verifying the correcteness of resilience properties during the operational phase of a system should be a requirement in critical infrastructures, including utility networks and industrial control systems, where services must be provided in an uninterrupted manner. Therefore, motivated by the absence of a practical approach, we examine in this paper how resilience policies can be specified and verified in the context of an actual access control model. For this purpose, the Attribute-Based Access Control (ABAC) was selected amongst others due to its flexibility and high level of expressiveness [15]. We anticipate a practical approach that would be able to efficiently ensure the resilience of policies. This will eventually reduce the overall complexity required for verifying resilience policies. The latter requirement derives mostly from the need to apply such a process in the operational phase of a system. Such functionality is currently absent from existing approaches since, firstly, the majority of solutions appear to propose the problem be solved during the design phase of a system, and secondly, tools that would help to facilitate the process of verifying resilience are absent from existing solutions, to the best of our knowledge. Hence, for us to achieve these objectives, we specify resilience policies in the context of ABAC, and embrace an existing set of tools, viz. the Access Control Policy Tool (ACPT) by NIST[1] and the NuSMV symbolic model checker[2] in order to specify ABAC-based policies and formally verify them, respectively [16].

The structure of the remainder of this paper is as follows: in Section 2 we elaborate on the relation between access control and resilience. A formal definition of our proposed ABAC model is provided in Section 3. Section 4 provides prerequisite information on the formal verification of policies in ABAC; and, resilience policies are specified and verified in Section 5. Concluding remarks and future work are discussed in Section 6.

2. ACCESS CONTROL AND RESILIENCE

In the context of industrial control systems cyber-security, resilience is a particularly significant issue. Such systems control physical processes, often safety-critical processes, in real time, in chemical production plants, water treatment facilities, nuclear power generation installations, oil and gas facilities and so on. Losses of availability and integrity in particular can have immediate and possibly unrecoverable effects. Such systems must simultaneously exclude adversarial intervention and ensure legitimation intervention. Thus access control always has to satisfy the dual requirement of denying access to certain types of actor but guaranteeing access for others. Following other authors [21], and our own prior work [13], the second requirement is labelled as *'resilience'*. As this second requirement has to be met at the same time as the first, and the two requirements must not contradict each other, it makes sense to express access control and resilience requirements using the same basic formalism, and to find a way of verifying them under some integrated mechanism.

With regards to access control in critical infrastructures – the authors in [22] provide information on how role based access control policies may appear in SCADA systems, and argue on the fact that roles and type of access on critical resources have to be clearly defined. A subset of roles in the hierarchy described in [22] is: *'Junior operator'*, *'Senior operator'*, *'Supervisor'*, and *'Manager'*. A set of operations is assigned with each role. Briefly, a user assigned with the *'Junior operator'* role, has a very restrictive set of operations, such as monitoring screens only; the *'Senior operator'* role offer operations such as these of starting or stopping a system and the potential to acknowledge an alarm (on top of the roles inherited by the *'Junior operator'* role); and, the *'Supervisor'* role offers the operation of disabling alarms (on top of the roles inherited by the *'Senior operator'* role) – a *'Manager'* is considered to have no restrictions, and thus can perform all the above operations [22], and also can be connected with other role hierarchies. In a scenario where the operations of starting, stopping a SCADA system and disabling alarms are considered to be critical, we must ensure the presence of personnel that would own the appropriate set of permissions to accomplish these critical operations successfully. Thus, in case of assigning *'user1'* with the role of *'Supervisor'* and *'user2'* with the role of *'Manager'* this policy can be characterised as being *'resilient'* since upon the absence (or removal) of one user, there still exist one disjoint set of users that contains one user authorised for starting, stopping the SCADA system and to acknowledge alarms.

With regards to resilience – this concept is identified to be of vital importance for organisations since it ensures an organisation's survivability and prosperity [5]. One of the important processes of resilience at an organisational level is operational resilience management, which in general refers to the set of strategies that when applied are able to protect and sustain the services and assets of an organisation [23]. Access management consists one of the operations introduced in an operational resilience management strategy. Its purpose is to ensure that the access granted to the subjects of a system (i.e. assignment with organisational assets) has to be proportionate with the business and resilience requirements [23]. Hence, to fulfil the resilience requirements, the subjects of a system (e.g. users) have to have a sufficient, yet not excessive, level of access to the organisation's assets. In order to successfully achieve this goal, a series of practices needs to be applied. Such practices, as identified by major research and development centres [23], are related with (i) enabling access, (ii) managing changes to access privileges, (iii) performing periodic review and maintenance of access privileges, and (iv) correction of inconsistencies.

In this paper, we investigate the first practice, i.e. enable access, that should be used in industrial control systems to ensure their protection and orderly functionality. The

[1]http://csrc.nist.gov/groups/SNS/acpt/
[2]http://nusmv.fbk.eu/

practice of enabling access is concerned with ensuring that the appropriate level of access to organisational assets is informed by resilience requirements [23]. This is of vital importance in critical infrastructures since their operational environment is required to keep access control current and reflective of the security and resilience requirements towards maximizing their availability [17].

3. ATTRIBUTE BASED ACCESS CONTROL MODEL

Attribute-based access control (ABAC) has gained the attention of researchers because it offers a high level of flexibility – it can implement various policies, and also it is an ideal candidate for use in highly-distributed and rapidly changing environments [15]. In general, access decisions in ABAC are based on the requester's owned attributes. The advantage of this approach is that it is possible to provide access to users in a collaborative environment without the need for them to be known by the resource a priori. This results in an inherent support for distributed access control and collaboration amongst domains. An implementation of ABAC can be seen in the eXtensible Access Control Markup Language (XACML) that is an OASIS standard[3]. And, another implementation of ABAC can be found in the Next Generation Access Control standard in [1].

In the following, we provide a definition of the ABAC model. This includes a reference to the main elements that could take part in the authorisation process, and a high-level description of its main administrative operations and administrative review functions.

3.1 Proposed Model

The definition of our ABAC model is based on the recommendations proposed by NIST in [14], where a set of guidelines forms the basis of a formal definition of ABAC. Thus, we provide all the required information with regard to the specifications of ABAC. Specifically, we elaborate on its main elements and the relation between them; we provide a formal definition of the model, and provide a list of ABAC's system and administrative functional specifications.

3.2 Elements

The ABAC model consist of the following six categories of elements: attributes, subjects, objects, operations, policies, and environmental conditions. A major difference between ABAC and other access control models is that in ABAC access is not granted or not based on the subject's identity, but rather it is evaluated on the basis of a set of attributes assigned to subjects and objects, as well as on environmental conditions. Figure 1 illustrates the main elements in ABAC and the interactions amongst them.

Attributes are characteristics of the subject, object, or environment conditions. Attributes may contain information given by a name-value pair, i.e. a tuple of the form: $(NAME, VALUE)$. As depicted in Figure 1, both subject and object attributes are able to support the use of meta-attributes. The latter provides an additional index for referring to groups of subjects and objects per se. Hierarchies in ABAC are intrinsically supported via the meta-attribute

[3]http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.html

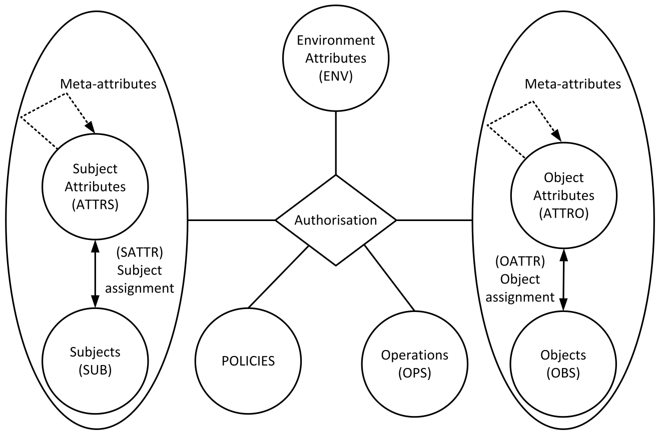
Environment Attributes (ENV)
Meta-attributes
Subject Attributes (ATTRS)
(SATTR) Subject assignment
Subjects (SUB)
Authorisation
POLICIES
Operations (OPS)
Meta-attributes
Object Attributes (ATTRO)
(OATTR) Object assignment
Objects (OBS)

Figure 1: ABAC's main elements and relationships

functionality. This provides ABAC with the potential to express powerful hierarchies between elements of the same type.

A **subject** is usually interpreted as being a user or process that issues access requests to perform operations on objects. Subjects can be assigned with one or more attributes.

An **object** can be a system resource for which access is managed by the attribute-base access control system. These could be devices, files, records, tables, processes, programs, networks, or domains containing or receiving information. It can be the resource or requested entity, as well as any entity on which an operation may be performed by a subject including data, applications, services, devices, and networks.

An **operation** is the execution of a function at the request of a subject upon an object. Example of operations include the read, write, edit, delete, copy, execute, and modify commands.

A **policy** is the representation of rules or relationships that makes it possible to determine if a requested access should be allowed, given the values of the attributes of the subject, object, and possible environment conditions.

An **environment condition** is an operational or situational context in which access requests occur. Environment conditions are detectable environmental characteristics. Environment characteristics are independent of subject or object, and may include the current time, day of the week, location of a user, the current threat level, etc.

The provision of the above definitions subsequently helps in the provision of a reference model for ABAC and a formal specification of it. In the following, we provide information about the main elements of the model and the relations between them.

3.3 Definitions

A formal model of the ABAC is defined as follows:

- SUB, OBJ, $ATTR_S$, $ATTR_O$, ENV, OPS, $POLICIES$ consist of subjects, objects, subjects' attributes, objects' attributes, environmental conditions, operations, and policies, respectively.
- $ATTR_S, ATTR_O, ENV$ are sets of subject, object and environmental conditions attributes in tuples of the form: $(NAME, VALUE)$.
- SUB is a set of tuples of the form: $(NAME, VALUE)$.

- OBJ is a set of tuples of the form: $(NAME, VALUE)$.
- $SATTR \subseteq (SUB \times ATTR_S)$ is a set of SUB and $ATTR_S$ mapping relation pairs.
- $OATTR \subseteq (OBJ \times ATTR_O)$ is a set of OBJ and $ATTR_O$ mapping relation pairs.
- $AssignedSubjects(a : ATTR_S) = 2^{SUB}$, or formally defined:
  $AssignedSubjects(a) = \{s \in SUB \mid (s, a) \in SATTR\}$.
- $AssignedObjects(a : ATTR_O) = 2^{OBJ}$, or formally defined:
  $AssignedObjects(a) = \{o \in OBJ \mid (o, a) \in OATTR\}$.
- OPS is a set of tuples of the form: $(NAME, VALUE)$.
- $POLICIES \subseteq SATTR \times OATTR \times ENV \times OPS$, where $POLICIES$ is a set of rules or relationships that makes it possible to determine if a requested access should be allowed, given the values of the attributes of the subject, object, and possibly environment conditions.

3.4 ABAC system and administrative functional specifications

The ABAC system and administrative functional specifications describe the main features required by an ABAC system. This includes the specification of a set of administrative operations and administrative review functions. The former consists of a set of functions that are required to administer the main elements of the access control model. These include operations such as the creation and deletion of elements, and assignments. The administrative review functions are capable of performing query operations on ABAC elements and relations. Tables 1 and 2 in Appendix B provide the function prototypes of the proposed ABAC model, including a short description of their functionality – these have been specified using a subset of Z notation, which is standardised in ISO/IEC 13568:2002 [18], but a full description of them is omitted here since it is out of the scope of this paper.

4. PRELIMINARIES ON VERIFICATION

An authorisation mechanism may include various complex operations, viz. assembles the policy, attributes and renders a decision based on the logic provided in the policy [14]. In this section, we provide information regarding some of the basic principles in temporal logic, which we use to specify policies in ABAC, and thus express authorisations. For more information, we refer the reader to [3]. The use of temporal logic, apart from providing a language for the property specification of policies, will eventually underpin the mathematical foundation used to formally verify authorisation policies. This requires the definition of a language for expressing polices and a transition system able to describe the behaviour of the access control model, and thus for properties to be verifiable for the model.

We consider AP to be a set of atomic propositions, and α, β and γ elements of AP. The set of propositional logic formulae over AP is inductively defined as:

- *true* is a formula;
- Any atomic proposition, which is element of AP is a formula;
- If Φ, Φ_1 and Φ_2 are formulae, then are $(\neg\Phi)$ and $(\Phi_1 \wedge \Phi_2)$;
- Nothing else is a formula.

We have that the conjunction operator $\wedge$ binds stronger then the derived binary operators, such as that of disjunction, implication, etc. Specifically, we define the former two as in the following: $\Phi_1 \vee \Phi_2 := \neg(\neg\Phi_1 \wedge \neg\Phi_2)$ and $\Phi_1 \rightarrow \Phi_2 := \neg\Phi_1 \vee \Phi_2$, respectively. The $\rightarrow$ means *'imply'*.

We also assume the following notation regarding the associativity and commutativity law for disjunction and conjunction: $\bigwedge_{1\leq i\leq n} \Phi_i$ for $\Phi_1 \wedge \ldots \wedge \Phi_n$ and $\bigvee_{1\leq i\leq n} \Phi_i$ for $\Phi_1 \vee \ldots \vee \Phi_n$. If $I = \emptyset$, then $\bigwedge_{i\in\emptyset} \Phi_i := true$ and $\bigvee_{i\in\emptyset} \Phi_i := false$.

Furthermore, we consider the *evaluation* of atomic propositions. This is done by assigning a truth value to each of them, i.e. a function $\mu : AP \rightarrow \{0, 1\}$, where 0 is *false* and 1 is *true*. The $\rightarrow$ means *'maps to'*. Therefore, a satisfaction relation $\models$ indicates the evaluations μ for which a formula Φ is *true*. Formally, it is written as:

- $\mu \models true$
- $\mu \models \alpha \Leftrightarrow \mu(\alpha) = 1$
- $\mu \models \neg\Phi \Leftrightarrow \mu \not\models \Phi$
- $\mu \models \Phi \wedge \Psi \Leftrightarrow \mu \models \Phi$ and $\mu \models \Psi$

Further on, we define the authorisation rule, property, and transition system of the ABAC model. The definitions are based on [14], but modified to fit the requirements of ABAC. Here we use the Computation Tree Logic (CTL) in order to specify policy properties. Linear-time Temporal Logic (LTL) could alternatively be used since we do not take advantage of the different expression level of neither CTL or LTL in our defined properties [19].

With regard to CTL, the prefixed path quantifiers assert arbitrary combinations of linear-time operators. Hence, we use the universal path quantifier $\forall$ that means *'for all paths'*, and the linear temporal operators $\Box$ and $\Diamond$ that mean *'always'* and *'eventually'*, respectively. Furthermore, we use the temporal modalities $\forall\Box\Phi$ representing *invariantly* Φ, and $\forall\Diamond\Phi$ representing inevitably Φ, where Φ is a state formula.

Definition 1. An ABAC rule is an implication of type '$c \rightarrow d$', where constraint c is a predicate expression as $(sub \wedge sattr \wedge obj \wedge oattr \wedge env \wedge ops)$, which when *true* implies the permission decision d. The $\rightarrow$ means *'imply'*.

Definition 2. An ABAC access control property p is an implication formula of type '$b \rightarrow d$', where the result of the access permission d depends on *quantified* predicate b on ABAC attributes and system states.

Definition 3. A transition system TS_{ABAC} is a tuple (S, Act, δ, i_0) where

- S is a set of states, $S = \{Permit, Deny\}$;
- Act is a set of actions,
  where $Act = \{(sub \wedge sattr \wedge obj \wedge oattr \wedge env \wedge ops), \ldots\}$ and $sub \in SUB$, $sattr \in ATTR$, $obj \in OBJ$, $oattr \in OATTR$, $env \in ENV$ and $ops \in OPS$;

- δ is a transition relation where $\delta : S \times Act \rightarrow S$;
- $i_0 \in S$ is the initial state.

The p in Definition 2 is expressed by the proposition $p : S \times Act^2 \rightarrow S$ of TS_{ABAC}, which can be collectively translated in terms of logical formula such that $p = (s_i \wedge \bigvee_{1 \leq i \leq n} (sub_n \wedge sattr_n \wedge obj_n \wedge oattr_n \wedge env_n \wedge ops_n)) \rightarrow d$ where $p \in P$ is a set of properties.

The behaviour of the system is defined by the ABAC rules, and they function as the transition relation δ in TS_{ABAC}. Thus, by representing an access control property using the temporal logic formula p, we can assert that model TS_{ABAC} satisfies p by $TS_{ABAC} \vDash \forall \Box (b \rightarrow \forall \Diamond d)$. Property $\forall \Box (b \rightarrow \forall \Diamond d)$ is a response pattern such that d responds to b globally (b is the cause and d is the effect) [25].

With regard to computational complexity – it is interesting to reference the computational complexity of the *'resiliency checking problem'*, which is NP-hard in the general case [21], and the computational complexity of model checking, which is P-complete for CTL and PSPACE-complete for LTL [3, 20]. However, we have to clarify that the computational complexity of the *'resiliency checking problem'* and that of verification of resilience specifications using model checking are not directly comparable. This is because in the former case a resilience policy is prepared from scratch taking into consideration resilience requirements, whereas in the latter case, resilience specifications are verified against an existing policy. We argue that in real-world cases, an initial set of resilience access control policies may be present and that they can change over time. Nevertheless, the development of a new – from scratch – resilience policy in operational environments is not always feasible due to operational requirements. Thus, the verification of resilience specifications may appear to be a more realistic and efficient solution.

5. VERIFICATION OF RESILIENCE POLICIES

5.1 Specification of resilience

In this section, we elaborate on the notion of resilience policies, and discuss how this could be interpreted in the context of the defined ABAC model. In order to do this, we embrace the definition of resilience policies defined in [21]. Specifically, a resilience policy is defined as the tuple of $ResiliencePolicy\langle P, s, d, t\rangle$, where P is the set of permissions, $s \geq 0$, $d \geq 1$ and $t \in N^+$ or $t = \infty$. Thus, a resilience policy is satisfied in an access control state *'if and only if upon removal of any set of s users, there still exist d mutually disjoint sets of users such that each set contains no more than t users and the users in each set together are authorised for all permissions in P'* [21]. The construction of a resilience policy is also known in the literature as the *'resiliency checking problem'* [8], [21]. Specifically, given a resilience policy tuple $ResiliencePolicy\langle P, s, d, t\rangle$ the solution provides an answer to the existence of binary relation between users U and permissions P, i.e. $UP \subseteq U \times P$ [21], or between users U and their authorised resources R, i.e. $UR \subseteq U \times R$ [8]. In general, permissions are considered to be operations on objects. Assuming the example in Section 2, we have the following critical operations on a SCADA system: *'monitor screen'*, *'start system'*, *'stop system'*, *'disable alarm'*, and *'change set points'*, and thus we set $P = \{$*Supervisor, Manager*$\}$. This is because $OATTR \times OPS$ is an ordered set that represents permissions P, and $ATTR_S \times P$ is also an ordered set that can be used for creating pairs of roles with permissions. Since both roles in the example are paired with all permissions, we can continue with the assumption that it is safe to use *roles* $\in ATTR_S$ and *permissions* $\in P$ interchangeably. Given P, we may have the following values for the rest of the resilience policy parameters: $s = 1$, $d = 1$, and $t = 1$. Specifically, $s = 1$ indicates that we want the policy to be resilient to the absence of any (one) user, $d = 1$ indicates that we require one set of users such that users in that set together possess all permissions; and, $t = 1$ since there is a single user that has all the permissions [21].

The definition of a resilience policy requires initially a careful definition of the different critical tasks in an organisation and subsequently identification of the main users and assigned permissions required to successfully complete these tasks. As mentioned already, this process can be performed during the early stages of the design of a system. Nevertheless, users and policies may change in a system, i.e. certain policies may be altered, deleted or new policies may be introduced. Therefore, these operations may introduce disruptions in an already existing resilience policy. Designing these policies from scratch may not be a viable solution, especially in the context of critical infrastructures, where systems must operate in an uninterrupted manner. Hence, administrators or operators in such environments may require to verify at any time the resilience offered by the active set of policies in their operational environment. Such an approach may also lead to reducing the overall complexity imposed by solving the resiliency checking problem from scratch.

Towards providing a viable solution to this requirement, we outline a process that is able to verify the resilience of a subset of ABAC policies. The resiliency checking problem is known to have various levels of complexity, introduced by the variance of each of the elements in the resilience policies tuple [21]. In this paper, we are concerned with the verification of resilience provided by a set of access control policies that are required to achieve a critical operation or task, and thus verify their resilience in the presence of several threats, e.g. absence of users that are responsible for completing - collaboratively or not - a specific critical operation or task.

In order to verify the resilience of ABAC policies, we use the response pattern as defined in Section 4. In general, we check resilience in ABAC policies between users and attributes – the latter representing permissions required to perform a task. For this, we use the satisfiability relation expressed by Formula 1.

$$TS_{RP\_ABAC} \vDash \forall \Box \Big(\bigwedge_{1 \leq i \leq n} !sub_n \bigwedge_{0 \leq i \leq m} attr_m \bigwedge_{1 \leq i \leq k} !sub_k \rightarrow \forall \Diamond Deny \Big) \tag{1}$$

where TS_{RP_ABAC} is the resilience ABAC policy transition system, sub_n, $sub_k \in SUB$, $sub_n \neq sub_k$, $attr_m \in ATTR_S : \{sub_n\} \times \{attr_m\} \in SATTR$, and $Deny \in S$ is the permission decision. In relation to the resilience policy tuple, i.e. $\langle P, s, d, t\rangle$, sub_n is mapped onto the set of users s that are considered to be absent; $attr_m$ refers to the attributes assigned with a user s and represent permissions required

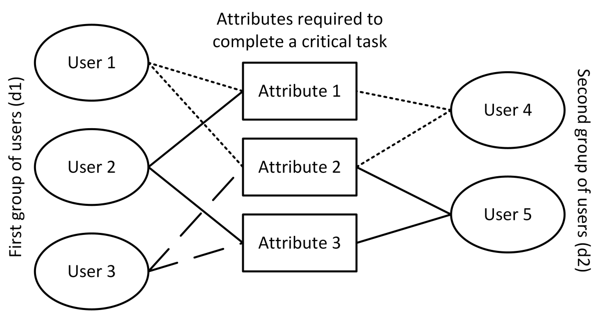
Attributes required to complete a critical task
First group of users (d1)
User 1
User 2
User 3
Attribute 1
Attribute 2
Attribute 3
User 4
User 5
Second group of users (d2)

Figure 2: Example of a resilience policy

to perform a task; and, sub_k refer to the mutual disjoint set of users expressed by d. With regard to the t parameter – this can be introduced implicitly by introducing additional specifications, following Formula 1.

5.2 Example

To demonstrate the applicability of the verification process in identifying resilience in ABAC policies, we choose to elaborate a generic, yet, representative example of a resilience policy, as described in [21]. Nevertheless, we also describe in Appendix A the implementation of an RBAC policy for a SCADA system using function calls of the proposed ABAC model (described in Appendix B).

The resilience policy in Fig. 2 assumes the existence of a critical task T. In order to successfully accomplish the critical task, the users (e.g. operators in a utility organisation) have to be collaboratively authorised for all three attributes. In this example, we consider two groups of users, where the first group includes the following users and assigned attributes: $User1 \times \{Attribute1, Attribute2\}$, $User2 \times \{Attribute1, Attribute3\}$, $User3 \times \{Attribute2, Attribute3\}$; and the second group includes the following users and attributes: $User4 \times \{Attribute1, Attribute2\}$, $User5 \times \{Attribute2, Attribute3\}$. In the context of an industrial control system, the above attributes could be equivalent with: $Attribute1 \equiv$ (monitor a device), $Attribute2 \equiv$ (start or stop a device), and $Attribute3 \equiv$ (maintain a device). Thus, in case of a device malfunction, operators (i.e. users of the system) shall be in position to monitor and acknowledge the problem, stop the faulty device, maintain the device, and finally, start the device. In order to define the above policy in ABAC and to formally verify its resilience, we use ACPT for the definition of access control policies and NuSMV for the verification of the resilience policy specifications.

In the following, we provide in Listing 1 the NuSMV code that defines the ABAC policy described in Figure 2.

Listing 1: *Specification of an ABAC policy in NuSMV*

```
MODULE main
VAR
    USER : {User1, User2, User3, User4, User5};
    ATTR : {Attribute1, Attribute2, Attribute3};
    ABAC_Policy01 : ABAC_Policy01(USER, ATTR);
ASSIGN
    next (USER) := USER;
    next (ATTR) := ATTR;
MODULE ABAC_Policy01(USER, ATTR)
VAR
    decision : {Permit, Deny};
ASSIGN
    init (decision) := Deny ;
    next (decision) := case
        USER = User1 & ATTR = Attribute1 : Permit;
        USER = User1 & ATTR = Attribute2 : Permit;
        USER = User2 & ATTR = Attribute1 : Permit;
        USER = User2 & ATTR = Attribute3 : Permit;
        USER = User3 & ATTR = Attribute2 : Permit;
        USER = User3 & ATTR = Attribute3 : Permit;
        USER = User4 & ATTR = Attribute1 : Permit;
        USER = User4 & ATTR = Attribute2 : Permit;
        USER = User5 & ATTR = Attribute2 : Permit;
        USER = User5 & ATTR = Attribute3 : Permit;
        1 : Deny;
    esac;
```

Subsequently, we define the set of specifications that are required to be verified on the transition system defined in Listing 1. We examine three different scenarios, where (i) we omit the policy introduced by group two; (ii) we omit the policy introduced by group one, and (iii) we consider the existence of both policies.

In the first scenario, we define two specifications in accordance with Formula 1. Therefore, making the assumption that $User1$ is absent, the CTL specifications that will verify the resilience of the examined policy are given in Listing 2. These specifications, when verified by the model checker, will provide a counterexample indicating which of the remaining users in group one (i.e. $User2, User3$) are in position to provide attributes $Attribute1$ and $Attribute2$, respectively. The verification of the given specifications and provision of counterexamples ensures the existence of resilience in the absence of $User1$. Specifically, it holds that $P = \{Attribute1, Attribute2, Attribute3\}$, $s = 1$, $d = 1$, and $t = 2$, with the latter stating that the set of users that together possess all permissions is equal to two.

In Listings 2 to 4, AG and AF are CTL expressions that are recognised by NuSMV as *'forall globally'* (i.e. $\forall \Box$) and *'forall finally'* (i.e. $\forall \Diamond$), respectively.

Listing 2: *Specification of resilience properties in CTL considering the absence of $User1$ and exclusion of the second group of users*

```
SPEC AG (( !(USER = User1) & (ATTR = Attribute1) &
          !(USER = User4) & !(USER = User5)
) -> AF decision =  Deny)
-- Evaluation: false, counterexample is provided

SPEC AG (( !(USER = User1) & (ATTR = Attribute2) &
          !(USER = User4) & !(USER = User5)
) -> AF decision =  Deny)
-- Evaluation: false, counterexample is provided
```

In the second scenario, we also define two specifications to verify the resilience of the examined policy in the absence of $User4$. In this case, the verification of the specifications provided in Listing 3 is evaluated as *true*. This is interpreted as: if $User4$ is absent then the remaining users (i.e. $User5$) do not collectively possess the required set of attributes to complete the critical task. No resilience is provided in this instance.

Listing 3: *Specification of resilience properties in CTL considering the absence of $User4$ and exclusion of the first group of users*

```
SPEC AG (( !(USER = User4) & (ATTR = Attribute1) &
           !(USER = User1) & !(USER = User2) &
           !(USER = User3)) -> AF decision =  Deny)
-- Evaluation: true

SPEC AG (( !(USER = User4) & (ATTR = Attribute2) &
           !(USER = User1) & !(USER = User2) &
           !(USER = User3)) -> AF decision =  Deny)
-- Evaluation: false, counterexample is provided
```

The third scenario under examination is presented in Listing 4. In this scenario, we assume the existence of both groups of users, and examine the resilience provided by the policy in the absence of $User5$. This is possible by omitting $\bigwedge_{1 \leq i \leq k} !sub_k$ in Formula 1. The evaluation of the defined specifications result in providing a countermeasure in both cases, and thus indicates the resilience of the policy. Specifically, in this instance, it holds that $P = \{Attribute1, Attribute2, Attribute3\}$, $s = 1$, $d = 2$, and $t = \infty$, with the latter stating that the set of users that together possess all permissions can be of any size.

Listing 4: *Specification of a resilience property in CTL considering both group of users*

```
SPEC AG (( !(USER = User5) & (ATTR = Attribute2)
) -> AF decision =  Deny)
-- Evaluation: false, counterexample is provided

SPEC AG (( !(USER = User5) & (ATTR = Attribute3)
) -> AF decision =  Deny)
-- Evaluation: false, counterexample is provided
```

6. CONCLUSION

In this paper, we examined an automated method for the formal verification of resilience specifications in the context of a specific access control model. For this purpose, we provided a formal definition of an ABAC model based on the guidelines provided by NIST; specified resilience using propositional logic; and, formally verified resilience specifications in a set of ABAC policies. We anticipate the research presented here will provide an interesting insight towards the consideration of resilience properties in addition to that of security in access control. The level of usability provided by existing approaches could be perceived as being low since they do not offer an appropriate set of tools that can automate the verification process. This holds mostly because existing approaches are considering the development of resilience policies during the design phase of a system. On the contrary, we have proposed the use of model checking as a means for the verification of resilience specifications in a set of existing policies during the operational phase of a system. Finally, by means of an example we demonstrated the applicability and level of automation offered by a computer-aided method such as model checking in verifying formally the correct functioning of ABAC policies against resilience specifications.

In future, we aim to investigate jointly the concepts of security and resilience in access control, including the possibility of conflicts that may arise.

7. ACKNOWLEDGEMENTS

This work is sponsored by the European Union under Grant SEC-2013.2.5-4: Protection systems for utility networks - Capability Project, Project Number: 608090, Hybrid Risk Management for Utility Providers (HyRiM).

8. REFERENCES

[1] ANSI. Information technology - Next Generation Access Control - Functional Architecture, 2013.

[2] V. Atluri and J. Warner. Supporting conditional delegation in secure workflow management systems. In *Proceedings of the tenth ACM symposium on Access control models and technologies*, pages 49–58. ACM, 2005.

[3] C. Baier, J.-P. Katoen, et al. *Principles of model checking*. MIT press Cambridge, 2008.

[4] E. Bertino, E. Ferrari, and V. Atluri. The specification and enforcement of authorization constraints in workflow management systems. *ACM Transactions on Information and System Security (TISSEC)*, 2(1):65–104, 1999.

[5] BSI. BS 65000 - Guidance for organizational resilience, 2014.

[6] D. Cohen, J. Crampton, A. Gagarin, G. Gutin, and M. Jones. Iterative plan construction for the workflow satisfiability problem. *Journal of Artificial Intelligence Research*, 51:555–577, 2014.

[7] J. Crampton, G. Gutin, and D. Karapetyan. Valued workflow satisfiability problem. In *Proceedings of the 20th ACM Symposium on Access Control Models and Technologies*, pages 3–13. ACM, 2015.

[8] J. Crampton, G. Gutin, S. Pérennes, and R. Watrigant. A multivariate approach for checking resiliency in access control. *arXiv preprint arXiv:1604.01550*, 2016.

[9] J. Crampton, G. Gutin, and R. Watrigant. Resiliency policies in access control revisited. In *Proceedings of the 21st ACM on Symposium on Access Control Models and Technologies*, pages 101–111. ACM, 2016.

[10] D. Ferraiolo, D. R. Kuhn, and R. Chandramouli. *Role-based access control*. Artech House, 2003.

[11] L. Gong and X. Qian. Computational issues in secure interoperation. *Software Engineering, IEEE Transactions on*, 22(1):43–52, 1996.

[12] A. Gouglidis, I. Mavridis, and V. C. Hu. Security policy verification for multi-domains in cloud systems. *International Journal of Information Security*, 13(2):97–111, 2014.

[13] A. Gouglidis, S. N. Shirazi, S. Simpson, P. Smith, and D. Hutchison. A multi-level approach to resilience of critical infrastructures and services. In *Proc. 23rd International Conference on Telecommunications (ICT 2016)*. IEEE, 2016.

[14] V. C. Hu, D. Ferraiolo, R. Kuhn, A. Schnitzer, K. Sandlin, R. Miller, and K. Scarfone. Guide to attribute based access control (ABAC) definition and considerations. *NIST SP*, 800:162, 2014.

[15] V. C. Hu, D. R. Kuhn, and D. F. Ferraiolo. Attribute-based access control. *IEEE Computer*, 48(2):85–88, 2015.

[16] V. C. Hu and R. Kuhn. Access control policy verification. *IEEE Computer*, 49(12):80–83, Dec 2016.

[17] Intel Security. Protect critical infrastructure. Technical report, McAfee. Part of Intel Security, 2016.

[18] ISO. IEC 13568: 2002: Information technology–Z formal specification notation–syntax, type system and semantics, 2002.

[19] R. B. Krug. CTL vs. LTL. Presentation, May 2010.

[20] F. Laroussinie, N. Markey, and P. Schnoebelen. Model checking CTL+ and FCTL is hard. In *International Conference on Foundations of Software Science and Computation Structures*, pages 318–331. Springer, 2001.

[21] N. Li, Q. Wang, and M. Tripunitara. Resiliency policies in access control. *ACM Transactions on Information and System Security (TISSEC)*, 12(4):20, 2009.

[22] M. Majdalawieh, F. Parisi-Presicce, and R. Sandhu. RBAC model for SCADA. In *Innovative Algorithms and Techniques in Automation, Industrial Electronics and Telecommunications*, pages 329–335. Springer, 2007.

[23] A. C. Richard, H. A. Julia, D. C. Pamela, W. W. David, and R. Y. Lisa. CERT resilience management model, version 1.0 improving operational resilience processes. Technical report, Software Engineering Institute, 2010.

[24] R. S. Sandhu and P. Samarati. Access control: principle and practice. *Communications Magazine, IEEE*, 32(9):40–48, 1994.

[25] SAnToS Laboraroty. Specification patterns, Responce property pattern, 2012.

[26] B. Shafiq, J. B. Joshi, E. Bertino, and A. Ghafoor. Secure interoperation in a multidomain environment employing RBAC policies. *Knowledge and Data Engineering, IEEE Transactions on*, 17(11):1557–1577, 2005.

[27] Q. Wang and N. Li. Satisfiability and resiliency in workflow authorization systems. *ACM Transactions on Information and System Security (TISSEC)*, 13(4):40, 2010.

APPENDIX

A. IMPLEMENTATION OF A RESILIENCE POLICY

In this section, we implement an RBAC resilience policy using the proposed ABAC model. Roles are introduced as subject attributes, and role hierarchy is supported via the meta-attribute functionality. The seniority of a role is expressed by the name-value tuples as *(senior role, junior role)*, with the senior role inheriting all the permissions of the junior role. In Listing A.1, we implement the RBAC policy defined in [22]. The offered resilience of this policy is easy to understand when examining the values of the $ResiliencePolicy\langle P, s, d, t\rangle$ tuple, i.e. $s = 1$, $d = 1$, $t = 1$, and $P = \{$*((action, Start system), (objectid, SCADA_DEV_001)), ((action, Stop system), (objectid, SCADA_DEV_001)), ((action, Acknowledge alarm), (objectid, SCADA_DEV_001)), ((action, Disable alarm), (objectid, SCADA_DEV_001))*$\}$, as explained in Section 5.1. Function calls used in Listing A.1 are briefly explained in Appendix B.

Listing A.1: *Implement an RBAC policy*

```
// Add new operations
 AddOperation(action, Monitor any screen);
 AddOperation(action, Start system);
 AddOperation(action, Stop system);
 AddOperation(action, Acknowledge alarm);
 AddOperation(action, Disable alarm);
 AddOperation(action, Change set point);
 AddOperation(action, Can change graphics);
 AddOperation(action, See alarm logs);
 AddOperation(action, Change security codes);
 AddOperation(action, Configure graphics);
 AddOperation(action, Controller setting);
 AddOperation(action, Security codes);

// OPS includes the following tuples
 OPS = {(action, Monitor any screen),
        (action, Start system),
        (action, Stop system),
        (action, Acknowledge alarm),
        (action, Disable alarm),
        (action, Change set point),
        (action, Can change graphics),
        (action, See alarm logs),
        (action, Change security codes),
        (action, Configure graphics),
        (action, Controller setting),
        (action, Security codes)}

// Add new subjects
 AddSubject(userid, user1);
 AddSubject(userid, user2);

// SUB includes the following tuples
 SUB = {(userid, user1), (userid, user2)}

// Add roles as new attributes
 AddAttribute(subject, (role, Junior operator));
 AddAttribute(subject, (role, Senior operator));
 AddAttribute(subject, (role, Supervisor));
 AddAttribute(subject, (role, Technician));
 AddAttribute(subject, (role, Engineer));
 AddAttribute(subject, (role, Manager));

// ATTRS includes the following tuples
 ATTRS = {(role, Junior operator),
          (role, Senior operator),
          (role, Supervisor), (role, Technician),
          (role, Engineer), (role, Manager)}

// Assign users with roles
 AssignSubject((userid, user1), (role, Supervisor));
 AssignSubject((userid, user2), (role, Manager));

// Introduce hierarchy relations (senior, junior)
// using the meta-attribute functionality
 AssignSubject((role, Supervisor),
               (role, Senior operator));
 AssignSubject((role, Senior operator),
               (role, Junior operator));
 AssignSubject((role, Engineer), (role, Technician));
 AssignSubject((role, Manager), (role, Supervisor));
 AssignSubject((role, Manager), (role, Engineer));
```

```
// SATTR includes the following tuples
 SATTR = {((userid, user1), (role, Supervisor)),
          ((userid, user2), (role, Manager)),
          ((role, Manager),
           (role, Supervisor)),
          ((role, Supervisor),
           (role, Senior operator)),
          ((role, Senior operator),
           (role, Junior operator))}

// Add new objects
 AddObject(objectid, SCADA_DEV_001);

// OBS includes the following tuples
 OBS = { (objectid, SCADA_DEV_001) };

// Policy definition
 AddPolicy(({}, (role, Junior operator)),
           ((objectid, SCADA_DEV_001), {}), {},
           (action, Monitor any screen));
 AddPolicy(({}, (role, Senior operator)),
           ((objectid, SCADA_DEV_001), {}), {},
           (action, Start system));
 AddPolicy(({}, (role, Senior operator)),
           ((objectid, SCADA_DEV_001), {}), {},
           (action, Stop system));
 AddPolicy(({}, (role, Senior operator)),
           ((objectid, SCADA_DEV_001), {}), {},
           (action, Acknowledge alarm));
 AddPolicy(({}, (role, Supervisor)),
           ((objectid, SCADA_DEV_001), {}), {},
           (action, Disable alarm));
 AddPolicy(({}, (role, Supervisor)),
           ((objectid, SCADA_DEV_001), {}), {},
           (action, Change set point));
 AddPolicy(({}, (role, Technician)),
           ((objectid, SCADA_DEV_001), {}), {},
           (action, Can change graphics));
 AddPolicy(({}, (role, Technician)),
           ((objectid, SCADA_DEV_001), {}), {},
           (action, See alarm logs));
 AddPolicy(({}, (role, Technician)),
           ((objectid, SCADA_DEV_001), {}), {},
           (action, Change security codes));
 AddPolicy(({}, (role, Engineer)),
           ((objectid, SCADA_DEV_001), {}), {},
           (action, Configure graphics));
 AddPolicy(({}, (role, Engineer)),
           ((objectid, SCADA_DEV_001), {}), {},
           (action, Controller setting));
 AddPolicy(({}, (role, Engineer)),
           ((objectid, SCADA_DEV_001), {}), {},
           (action, Security codes));
```

B. ADMINISTRATIVE OPERATIONS AND REVIEW FUNCTIONS IN ABAC

Tables 1 and 2 include a list of administrative operations and review functions respectively of the proposed ABAC model. Although not all of them are demonstrated in this paper, we include them for completeness.

Table 1: **Administrative operations in ABAC**

Operation	**Description**
AddSubject (subject?: SUB)	Creates a new subject
DeleteSubject (subject?: SUB)	Deletes an existing subject from the ABAC database
AddObject (object?: OBJ)	Creates a new object
DeleteObject (object?: OBJ)	Deletes an existing object from the ABAC database
AddOperation (operation?: OPS)	Creates a new operation
DeleteOperation (operation?: OPS)	Deletes an existing operation from the ABAC database
AddAttribute (type?: subject | object, attr?: $ATTR_S \mid ATTR_O$)	Creates a new attribute
ModifyAttribute (type?: subject | object, attr?: $ATTR_S \mid ATTR_O$)	Modifies the value of an existing attribute
DeleteAttribute (type?: subject | object, attr?: $ATTR_S \mid ATTR_O$)	Deletes an existing attribute from the $ATTR$ set
AddEnvironment (attr?: ENV)	Creates a new environment condition attribute
ModifyEnvironment (attr!: ENV, value?: VALUE)	Modifies the value of an existing environment attribute
DeleteEnvironment (attr?: ENV)	Deletes an existing attribute from the ENV set
AssignSubject (subject?: SUB, attr?: $ATTR_S$)	Assigns a subject to an attribute
DeassignSubject (subject?: SUB, attr?: $ATTR_S$)	Deassigns a subject from an attribute
AssignObject (object?: OBJ, attr?: $ATTR_O$)	Assigns an object to an attribute
DeassignObject (object?: OBJ, attr?: $ATTR_O$)	Deassigns an object from an attribute
AddPolicy (sattr?: SATTR, oattr?: OATTR, env?: ENV, ops?: OPS)	Adds an action to a subject to perform an operation on an object given the subject's and object's attribute values, and potential environment attribute values
DeletePolicy (sattr?: SATTR, oattr?: OATTR, env?: ENV, ops?: OPS)	Deletes the action from a subject to perform an operation on an object given the subject's and object's attribute values, and potential environment attribute values

Table 2: **Administrative review functions in ABAC**

Function	**Description**
AssignedSubjects (attr?: $ATTR_S$, result!: 2^{SUB})	Return the set of subjects assigned to an attribute
AssignedObjects (attr?: $ATTR_O$, result!: 2^{OBJ})	Return the set of objects assigned to an attribute
SubjectAttributes (sub?: SUB, result!: 2^{ATTR_S})	Return the set of attributes assigned to a subject
ObjectAttributes (obj?: OBJ, result!: 2^{ATTR_O})	Return the set of attributes assigned to an object
SubjectAllAttributes (sub?: SUB, result!: 2^{ATTR_S})	Return the set of all attributes a subject may be eligible for, including attributes inherited from potential hierarchies implemented using the meta-attribute functionality
ObjectAllAttributes (obj?: OBJ, result!: 2^{ATTR_O})	Return the set of all attributes an object may be eligible for, including attributes inherited from potential hierarchies implemented using the meta-attribute functionality