

A Policy Framework for Personalized and Role-Based SPIT Prevention

Nico d'Heureuse
NEC Europe
Kurfürsten-Anlage 36
Heidelberg, germany
dheureuse@nw.neclab.eu

Jan Seedorf
NEC Europe
Kurfürsten-Anlage 36
Heidelberg, germany
seedorf@nw.neclab.eu

Saverio Niccolini
NEC Europe
Kurfürsten-Anlage 36
Heidelberg, germany
niccolini@nw.neclab.eu

ABSTRACT

Voice over IP (VoIP) deployment is increasing at fast pace. Due to the expected decrease of cost for call initiations with VoIP and the risk of infected devices, Spam over IP Telephony (SPIT) is likely to be a serious threat for VoIP service architectures in the near future. Since SPIT is a very personal matter, users must be able to express the level of intrusiveness acceptable to them when receiving calls, i.e., the degree to which a callee is willing to be disturbed by potentially unsolicited calls. Further, companies have a need to enforce *role-based* and *status-based* SPIT protection policies in order to allow, e.g., a higher level of intrusiveness for a secretary during the day than for the CEO on his/her mobile at night.

In this paper, we derive requirements for a protection system that enables personalized and role-based SPIT prevention. We examine existing solutions and show that they are insufficient to meet these requirements. Based on this comparison, we design a framework for personalized SPIT prevention. To demonstrate that our framework is capable of meeting the requirements, we give examples that show how important use cases can be addressed with this framework. Finally, we report on our prototypical implementation of the framework in a SIP PBX.

Categories and Subject Descriptors

C.2.0 [Computer-Communication Networks]: General—*Security and protection*; K.6.5 [Management of Computing and Information Systems]: Security and Protection

General Terms

Security

Keywords

Personalization, Authorization policies, Spam Over IP Telephony (SPIT), VoIP Security

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

IPTCOMM '09, July 7-8, 2009, Atlanta, Georgia, USA
Copyright 2009 ACM X-XXXXX-XX-X/XX/XX ...\$5.00.

1. INTRODUCTION

Voice over IP (VoIP) is slowly replacing the Public Switched Telephony Network (PSTN). While this new technology offers cost savings and other advantages such as easy integration with web-services, security for VoIP remains a concern. Since the cost for each call initiation is expected to decrease with VoIP compared to the PSTN, this technology is likely to attract spammers. Further, since VoIP entities are by definition IP-based devices, VoIP terminals and servers are vulnerable to many attacks common in today's Internet [1]. This makes VoIP an even more attractive target for spammers because it enables for spam to be sent from trojanized devices (e.g., infected VoIP terminals belonging to a botnet controlled by a spammer). Therefore, Spam over IP Telephony (SPIT) is to be considered a serious threat for VoIP service architectures.

While email spam is a well-known threat in today's Internet, SPIT is relatively new and has some key differences to email spam. For instance, a SPIT-call immediately disturbs the user with a ringing phone. Further, content filtering (which is very successful in filtering email-spam) of real-time VoIP calls is computationally very hard and not much help against SPIT since there is no content available until the call *has already been* accepted by the callee. Thus, there is a need for new approaches for protection against this threat.

Researchers have started to address this problem and proposed various mechanisms which strive to detect unsolicited VoIP calls. So-called *non-intrusive* anti-SPIT mechanisms analyze the messages transmitted during a VoIP call without interacting with the caller nor the callee. Examples for such methods are blacklisting, whitelisting, as well as analysis of call patterns, call rates, or similar statistical values [2]. Mechanisms *interacting with the caller* impose a certain challenge to the initiator of a call that needs to be solved in order to reach the callee. Examples for such mechanisms are hash-cash [9] (where the calling entity can be requested to compute a computationally expensive function in order to reach the callee), voice-energy level comparison [14] (where the caller is greeted with a message and it is measured if the caller is quiet during this greeting message), or more generally CAPTCHA-tests¹ [20]. In addition, researchers have proposed mechanisms that *interact with the callee of a call*. For instance, in [15] consent-based communications have been proposed, where the callee of a VoIP call can decide to accept a call based on the identity of the caller. Another example for a SPIT protection method interacting

¹Completely Automated Public Turing Test to Tell Computers and Humans Apart

with the callee is feedback from the callee [12], where the callee can press a special hang-up button on its phone to signal to its proxy that the last call received was SPIT. In [13], a holistic approach has been presented for combining the various techniques against SPIT in an overall SPIT protection system. [13] uses the concept of a *SPIT score* (see also [23]): multiple protection mechanisms (e.g., statistical tests) contribute to an overall SPIT score, whereby high scores represent a high level of probability of the call being unsolicited.

In summary, a variety of SPIT prevention mechanism as well as approaches for combining the various methods into an overall protection system exist. It is, however, important to realize that SPIT is of a very personal and subjective nature: Some users may actually want to receive certain calls like advertisements while others may regard the same type of calls as annoying. Thus, each callee has its own acceptable level of *intrusiveness*, demanding for personalized protection settings. In addition, each user might want to have different protection settings depending on the time of day or presence status. Consider a manager who might want his/her SPIT protection system to be aligned with his/her outlook calendar, allowing for a very low level of intrusiveness during meetings and a reasonable level of intrusiveness otherwise. Hence, each user needs a flexible way of expressing his/her personal level of intrusiveness regarding incoming SPIT calls.

Another reason why a single, *global* SPIT prevention setting is of little value is the fact that in many real-world scenarios (e.g., in companies) users have different *roles*, and a single user can have several different roles with potentially conflicting policies. A company might enforce different SPIT protection policies for users in different *groups* (i.e., with different roles). Similar to access control, this demands for *role-based* policies for SPIT protection. We argue that any real-world SPIT protection system must be able to express role-based policies and in addition must be capable of resolving conflicting policies (e.g., if a user is member of a several groups, or if the user has set additional, personalized SPIT protection attributes).

In this paper, we address the issues raised above. Our goal is the design of a framework for SPIT prevention policies which enables to specify personalized and role-based SPIT prevention. By using such a policy framework, users and administrators can express the desired handling of incoming VoIP calls with respect to SPIT. The resulting policies are used to authorize specific VoIP entities (e.g., a SIP service provider or a local IP PBX in a company) to handle incoming calls accordingly. We derive requirements for a *parametrizable* (i.e., combining different protection mechanisms with different settings), *role-based* (i.e., allowing for different levels of intrusiveness depending on the role of a user), and *personalized* (i.e., allowing each user to specify the individually desired level of intrusiveness) SPIT prevention policy framework. We show that existing approaches are insufficient to meet these requirements. Consequently, we present the design of a role-based personalization framework for SPIT prevention, reusing existing approaches when possible. We have implemented this framework and give some examples on how important use cases can be addressed with our framework.

The rest of this paper is organized as follows. In section 2 we derive requirements for a SPIT prevention policy

framework. We survey existing approaches in section 3 and compare these approaches with our requirements, showing where they are insufficient. In section 4 we present and describe in detail our proposed SPIT prevention policy framework. Further, we give examples how important use cases can be addressed with our approach. Finally, we report on our prototypical implementation of our proposed framework in section 5. We conclude the paper with a summary and an outlook on future work in section 6.

2. REQUIREMENTS FOR A SPIT PREVENTION POLICY FRAMEWORK

To derive requirements for a policy framework², we consider the following real-world scenario: A company would like to deploy a SPIT prevention system for its VoIP infrastructure. With respect to employees, the company has defined certain roles (e.g., 'internship-student', 'secretary', 'manager', 'member of the board', ...) which are already expressed in the company's access control policies (e.g., using Role-Based Access Control, RBAC [8]). Similarly, the company would like to have SPIT protection so that the probability of an employee being disturbed by unsolicited calls depends on the employee's role in the company. For instance, a sales office is likely to accept some SPIT calls rather than risking to miss an important customer call. In comparison, a back-office employer needs a higher level of protection against unsolicited calls so that he/she can concentrate on his/her work and is not disturbed frequently.

The reason why different levels of protection are necessary for different roles is that there is a tradeoff inherent in any kind of SPIT protection: Researchers agree that most probably there will be no SPIT prevention system which can guarantee a 0-percent false positive rate (i.e., never falsely marking a legitimate call as SPIT) and at the same time offer a 0-percent false negative rate³ (i.e., never falsely marking a SPIT call as legitimate) [17]. In other words, any kind of system needs to be configured (e.g., by combining algorithms and through input parameters) to achieve the desired level of intrusiveness for its users. In general, different levels of intrusiveness are desired for different types of employees. Thus, depending on the telephony use case of the employee, different protection settings (and consequently different settings and combinations of SPIT-protection algorithms) are suitable for different roles.

Moreover, it shall be possible for individual employees to

²Requirements for authorization policies to be used for preventing SPIT have also been presented in [19]. In principle, the requirements in [19] are similar to our requirements but on a more detailed, fine-grained level. In contrast, our requirements are rather high-level in order to highlight the general capabilities necessary for a SPIT prevention policy framework. In addition, we deduce our requirements from a real-world scenario to motivate them for readers not familiar with research in the field of SPIT prevention, and to make this paper self-contained.

³The fact that there is no 100-percent protection against SPIT has also legal implications. In most countries providers are only allowed to automatically filter out calls which they can clearly identify as malicious. Thus, they are legally not allowed to automatically filter potential SPIT calls without the callee's permission. This is another reason why authorization policies are needed: These policies can be used to explicitly have the callee permit VoIP entities (e.g., the callee's VoIP provider) to block certain types of calls.

specify personal and status-based demands. For example, a manager can direct a higher percentage of suspicious calls to his/her mailbox during important meetings than on average. Similarly, a marketing department may want to whitelist certain identities in order to receive advertisements calls and analyze phone marketing strategies of competitors. However, when such personal settings are in a conflicting state with role-based policies, these conflicts must be resolved. For instance, some company-wide policies may be mandatory and it shall not be possible to circumvent (*overwrite*) these with personal user settings. Other role-based policies may simply be default settings which can be changed by individual employees. The system must be capable of expressing both cases, i.e., mandatory as well as standard, changeable settings.

To allow for flexible and extensible protection against SPIT, the company envisions a solution which can combine different SPIT protection algorithms, where each of these algorithms can be configured with (potentially) several parameters. This is a key requirement with respect to policies because, depending on the desired level of intrusiveness for a certain user (or group of users), certain protection modules with certain settings might be necessary while others may not be. For instance, some users may want an audio CAPTCHA to be executed in English on all incoming calls while others may want to have an audio CAPTCHA to be executed in German but only on calls with certain statistics. In other words, the possibility to express individual levels of intrusiveness requires a flexible, parametrizable SPIT prevention solution where the multiple different protection modules can be selected, parametrized, and combined through policies.

Furthermore, it can be assumed that some anti-SPIT mechanisms return not only a single test result but instead a set of result values. For instance, a voice CAPTCHA might return “`type=CAPTCHA, result=passed, numberOfRetries=0, language=en`” meaning that the CAPTCHA was passed upon the first try and that the message was played in English. In general, a policy framework must therefore enable to take more than a single test result value into account for fine grained customization.

In conclusion, within a single enterprise network many different users may prefer different individual SPIT protection settings. Additionally, various use cases for SPIT prevention arise due to the different roles of employees within a company. At the same time, company policies (enforced settings) must not be violated. Therefore, the key requirements that can be identified for a policy framework for SPIT prevention are the following:

- **Personalization:** The framework must be capable of keeping *per-user* settings with respect to SPIT prevention so that users can express their individually desired levels of intrusiveness.
- **Parametrizable Solution:** The framework must be able to support the combination and parametrization of different SPIT protection methods and it must be extensible so that new techniques (against newly appearing attack types) can be integrated. Only by parametrizing and combining different algorithms it is possible to reach different levels of intrusiveness as demanded by users.

- **Context-based Protection:** Users must be able to not only set individual settings but additionally to specify different SPIT protection settings depending on the current context or status.
- **Role-based Policies:** Multiple policy levels based on roles must be supported so that an administrator is able to set default and mandatory settings for certain roles.
- **Handling multi-valued test results:** The framework must be able to take into account multi-valued results from SPIT protection algorithms which produce more than one result value.
- **Resolving Policy Conflicts:** The framework must be able to resolve potentially conflicting policies. It must be able to specify which policies are mandatory and therefore must be enforced.

3. BACKGROUND AND EXISTING APPROACHES

Various policy languages have been proposed in the context of computer security, mostly for access control (see [5] or [7] for an overview). In this section we focus on surveying existing approaches for policies in the context of SPIT prevention. We describe such related work and discuss why it cannot satisfactorily fulfill our requirements.

The authors of [18] present a policy-based approach for SPIT prevention. The purpose of this approach is to precisely specify vulnerabilities in order to detect attacks on VoIP networks. This is a completely different use-case as the one we envision: Our goal is to enable *personalized* execution of *different* SPIT protection methods. In other words, [18] describes a single protection mechanism whereas we specifically consider the customized combination of multiple protection algorithms.

Several publications consider the problem of handling conflicting policies in the context of telecommunications and more specifically in the context of call control (e.g., [4]). Our work concerns the specification of suitable policies for call handling specifically in the context of SPIT prevention. Our framework enables to prioritize potentially conflicting policies and provides means to solve conflicts once they occurred. However, conflict detection and avoidance are not the main focus of our work. Therefore, we regard existing literature on policy conflicts for call control as very related to our work but concerning a separate problem.

3.1 Common Policy (RFC 4745)

RFC 4745 [16] defines an authorization policy framework for controlling access to application-specific data. The actual policies are specified using XML. The framework is very general and only defines very basic elements. However, it is designed to support domain specific extension. A policy is specified using a set of rules. Each rule consists of conditions and actions (which are executed if the conditions match). One very important concept of the framework is that all rules within a policy are treated equally, i.e., no rule takes precedence over another rule. The conditions of all rules are evaluated simultaneously, and all actions of all matching rules are executed. Since this can potentially lead to conflicting actions, the framework defines how these conflicts are resolved, i.e., how different actions are combined

(see section 10.2 of RFC 4745 and section 4.2.3 of this paper). As a general rule, RFC 4745 [16] suggests to always execute the least restrictive action.

A policy according to Common Policy can be specified using an XML document (see listing 1). Each XML document contains a set of rules (`<rule>`) within a `<ruleset>` element. Each rule consists of conditions (within a `<conditions>` element), actions (within a `<actions>` element), and transformations (within a `<transformations>` element).

`<conditions>` elements are used to specify which conditions must be met before the actions and transformations can be executed. Each type of condition specifies when it is met, i.e., when it evaluates to `TRUE`. A rule belongs to the matching rule set only if *all* its conditions evaluate to `TRUE`.

While the conditions describe the *if*-part of a rule, the actions and transformations define its *then*-part, i.e., the operations to be executed once all conditions are met. Each type of action or transformation specifies which operation is to be executed. Transformations are used when the accessed data must be modified; actions are used for all other types of operations.

The standard itself only specifies very basic conditions. These include methods for checking the identity of a user (`<identity>`) or for specifying the time period in which a rule should be valid in (`<validity>`). The standard does not specify any actions or transformations. However, rules are specified to combine multiple “permissions”, i.e., actions or transformations, with each other:

Each type of permission is combined across all matching rules. Each type of action or transformation is combined *separately and independently*. The combining rules generate a combined permission. The combining rules depend only on the data *type* of permission. [...] For integer [...] permissions, the resulting permission is the *maximum* value across the permission values in the matching set of rules.

Thus, considering two integer-type actions `<action>1</action>` and `<action>2</action>` the resulting action would be `<action>2</action>` since $\max(1, 2) = 2$.

```
<ruleset>
  <rule id="rule1">
    <conditions>
      ...
    </conditions>
    <actions>
      ...
    </actions>
    <transformations>
      ...
    </transformations>
  </rule>
  <rule id="rule2">
    <conditions>
      ...
    </conditions>
    <actions/>
    <transformations/>
  </rule>
</ruleset>
```

Listing 1: Syntax of Common Policy (RFC 4745)

3.2 SPIT Authorization Policy Extensions

In [21] the authors propose extensions to RFC 4745 [16] specifically targeted at SPIT prevention. The goal of this work is to include SPIT protection methods in the policy format to allow personalized SPIT protection. The work defines the following new extensions:

- Conditions
 - `<spit-handling>` and `<challenge>` subcondition: Used to evaluate the result of previously executed SPIT protection mechanisms. The `<spit-handling>` condition evaluates to `TRUE` if at least one of the `<challenge>` subconditions evaluates to `TRUE`.
 - `<sphere>`: Allows evaluating the current *sphere* of the user, e.g., “Office” or “Home”
 - `<presence-status>`: allows evaluating the current *presence status* of the user, e.g., “available”, “busy” or “in-meeting”
 - `<time-period>`: This element is an extension of the `<validity>` condition (see [16]) which allows making decisions depending on the time, date and timezone.
- Actions
 - `<execute>`: Used for executing a specific SPIT protection method.
 - `<forward-to>`: Used to forward a call to a different location.

The proposed extension is still in an early stage. It is therefore still quite inconsistent and has some shortcomings (see section 3.4).

3.3 Call Processing Language (CPL)

The Call Processing Language (CPL), defined by RFC 3880 [11], allows users to specify how incoming (or outgoing) calls should be processed using an XML document. In contrast to the rule evaluation of Common Policy [16], where each rule is evaluated independently of each other and the results are merged, the processing of a document with CPL is similar to the processing in a regular programming language. When compared to a full-fledged programming language, the features of CPL are, however, very limited. For instance, loops and jumps are not supported. These limitations were introduced on purpose to make sure that a CPL program will always terminate and will thus only use a limited amount of computing capacity on the servers.

Each CPL program represents an if-then-tree. The nodes of this tree represent the actions, e.g., the redirection of a call; the tree edges represent the evaluation of certain conditions, e.g., the time-of-day or the identity of the caller. This tree can efficiently be processed by computers and is easy to understand for humans.

One basic concept of CPL is extensibility, i.e., new conditions and actions can be added easily to CPL. E.g., in [10] the authors present an extension to use presence services for call control.

3.4 Shortcomings of Existing Approaches

Since CPL was specifically designed for personalized call routing, which is quite similar to applying different SPIT protection mechanisms, it is very related to our work. Extensions for applying and evaluating SPIT protection mechanisms could easily be added. However, the *tree*-nature of CPL renders it less useful for our purposes. One of our requirements (see section 2) is the support of role-based policies. In other words, we must be able to merge the policy documents of different roles in a consistent and deterministic manner. In the context of CPL we would have to merge multiple if-then-trees, which is a difficult task especially in the case of multi-level and role-based policies. Hence, we decided not to consider CPL as a possible basis for our envisioned framework.

Common Policy [16] is a good basis for our work because it provides the base primitives for the policy framework we envision. However, it is a general specification that does not define any actions or transformations. Moreover, it is not directly targeted at SPIT prevention.

The Internet-Draft [21] proposes extensions to Common Policy which are closely related to our work. In principle, the extensions proposed in [21] can achieve personalized SPIT prevention and the work provides initial solutions to the problem. However, the draft is in an early stage and there are still some inconsistencies within the specification (e.g., inconsistent naming of XML elements). Furthermore, in its current state, the proposal does not sufficiently define how possible action conflicts are solved (see section 4.2.3 for details). The authors of [21] also assume that each SPIT protection mechanism only returns a single result value, which is a strong (and probably unrealistic) restriction, contradicting with our requirements (see section 2). Finally, role-based policies – one of our key requirements – are not taken into account.

Nevertheless, whenever possible we reuse the concepts proposed in draft-tschofenig-sipping-spit-policy [21] in our framework. More specifically, we allow the usage of all conditions listed in section 3.2, with the exception of the `<challenge>` subcondition of `<spit-handling>`. We redefine this subcondition in section 4.2.2. The actions defined in [21] do not fulfill our requirements and are thus either dropped (`<forward-to>`) or redefined (`<execute>`, see section 4.2.3).

In summary, while there are existing approaches for personalized SPIT prevention, these are not sufficient to meet our requirements and they cannot address all use-cases for customizable SPIT prevention we assume to be realistic. In the following, we therefore present new Common Policy extensions (which enable to satisfy our requirements), reusing previous extensions when applicable.

4. PROPOSED FRAMEWORK

In this section we describe the design of our framework. Our approach is using and extending the basic elements of RFC 4745 [16]. Since [21] already suggests some extensions for personal SPIT policies based on RFC 4745, we re-use some of these extensions when applicable. For the use-cases and the requirements where [21] does not provide a solution (see section 3.4) we define new extensions. For distinction of the source of the different policy elements, in the remainder of this document we use XML namespaces as follows:

- `<cp:...>`: This element is defined in RFC 4745[16],

- `<spit:...>`: This element is defined in [21],

- `<spf:...>`: This element (`spf` for SPIT Policy Framework) has not been defined previously. It is a new element we use for our proposed extensions.

4.1 Introductory Example

The best way to quickly understand the concept of the proposed policy format is by looking at an example. Listing 2 shows such an example, containing two different rules. The example assumes that the call already has been analyzed by some SPIT rating mechanisms and has been assigned a SPIT-score. According to the concepts of RFC 4745 all rules are applied simultaneously and the actions for the matching rule set are executed. The first rule (“`allowLowScored`”) of Listing 2 matches only if there is a SPIT test result with name “`spitScore`” available which has an attribute “`totalScore`” whose value is less than (“`lt`”) 5. Upon a match, the action “`allow`” is executed, i.e., the call is forwarded to the user. The second rule (“`blockOthers`”) always matches since it has no conditions. The specified “`block`” action is thus executed for every call. In consequence, for every call with a SPIT score of less than 5 both actions, i.e., “`allow`” and “`block`”, are active. The action merging rules of RFC 4745 specify that in case of such an action conflict, the least restrictive action should be applied if no other merging rule is specified (see section 4.2.3 for an exact definition of the merging rules for our `<spf:execute>` action). In our example the “`allow`” action is, from the callers point of view, the less restrictive one (compared to blocking the call) and would thus be the resulting action. In general, the application enforcing a policy must have knowledge of the *restrictiveness* for all types of actions (e.g., by having an ordered list of possible actions for action type `<spf:execute>`). In summary, the policy defined in listing 2 forwards all calls with a SPIT score of less than 5 and blocks all other calls.

```
<cp:ruleset xmlns:cp="..."
  xmlns:spf="..."
  xmlns:spit="...">
  <cp:rule id="allowLowScored">
    <cp:conditions>
      <spit:spit-handling>
        <spf:challenge ref="spitScore">
          <spf:lt name="totalScore">5</spf:lt>
        </spf:challenge>
      </spit:spit-handling>
    </cp:conditions>
    <cp:actions>
      <spf:execute>allow</spf:execute>
    </cp:actions>
    <cp:transformations />
  </cp:rule>

  <cp:rule id="blockOthers">
    <cp:conditions />
    <cp:actions>
      <spf:execute>block</spf:execute>
    </cp:actions>
    <cp:transformations />
  </cp:rule>
</cp:ruleset>
```

Listing 2: Introductory Example

```

<cp:conditions>
  <spf:rule-level>5</spf:rule-level>
</cp:conditions>

```

Listing 3: Rule level example

4.2 New Extensions

In order to fulfill the requirements described in section 2, we introduce two new concepts: *rule levels* and *action priorities*. Rule levels enable to specify the order in which rules are evaluated. This allows setting mandatory policies for certain roles. Rule levels are realized with the new `<spf:rule-level>` condition. Action priorities enable to overwrite certain default settings.

In addition, we introduce the new `<spf:challenge>` condition which has a similar function as the `<spit:challenge>` condition defined in [21], but which is capable to process multiple return values per executed SPIT test. Finally, we specify a new `<spf:set>` transformation which enables to set individual parameters for executed actions.

4.2.1 Rule Level condition

To support the combination of policies for different user roles we introduce the new concept of *rule levels*.

The basic idea is to assign a certain numerical level to each rule. When evaluating the ruleset, rules with higher levels are taken into account only if the rules with lower levels did not result in a final action. This can be used to evaluate different sets of rules separately from each other (e.g., first evaluating company wide rules before personal settings are taken into account). System administrators could assign a certain range of rule levels to be used by the users, while other rule levels can only be used by the administration.

For specifying a rule level, we introduce the `<spf:rule-level>` condition. It evaluates to TRUE if the rule level which is currently being evaluated corresponds to the value given in the element's body. The policy decision point (PDP, see [22]) will start with evaluating the rules of level 1 only. If no final action can be determined, the level is increased by one and the rules are re-evaluated. The exact definition of what a *final* action is application specific. We assume that any `<spf:execute>` action (see section 4.2.3) is a final action. Listing 3 shows an example of a condition with rule level 5.

Note that, since the assignment of a certain level to a rule is done using a condition, the absence of this condition means that the rule will always be evaluated, i.e., if no rule level is explicitly specified, the rule belongs to all levels.

4.2.2 Challenge condition

A method is needed to evaluate the results of the SPIT protection procedures that have been applied.

draft-tschofenig-sipping-spit-policy [21] defines the `<spit:spit-handling>` condition and its `<spit:challenge>` sub-condition exactly for this purpose, i.e., to check the return value of a previously executed SPIT protection method. The `<spit:challenge>` element allows specifying only a test name (given as a test URI) and the expected result. Note that only a single test result is expected per SPIT test. Since our assumption is that a SPIT test might return more than one result value,

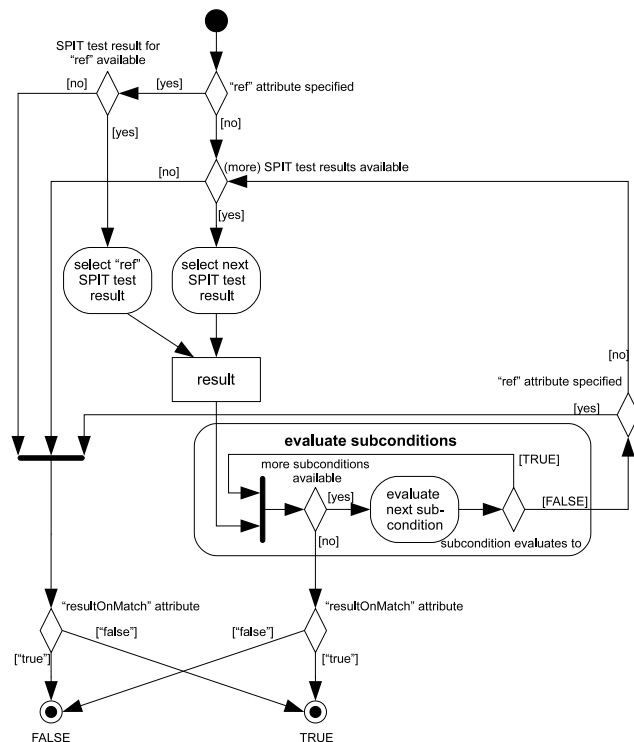


Figure 1: Evaluation procedure for the `<spf:challenge>` condition

`<spit:challenge>` does not meet our requirements.

We assume that each SPIT test can return a set of attribute-value pairs. Within the policy framework, we must be able to make decisions depending on any of the returned attribute-value pairs. The `<spit:challenge>` condition, as defined in draft-tschofenig-sipping-spit-policy [21], however, allows only checking a single value. We therefore redefine and extend the `<spit:challenge>` condition and define our new condition `<spf:challenge>` as follows:

The `<spf:challenge>` condition searches through the available SPIT test results using the subconditions (see below) specified within the condition's body. An attribute `resultOnMatch` with values "true" or "false" may be specified. The default value for the `resultOnMatch` attribute is "true". Furthermore, an optional `ref` attribute can be used to explicitly define which of the SPIT test results should be evaluated. Figure 1 shows a graphical representation of the evaluation process for the `<spf:challenge>` condition.

The `ref` attribute corresponds to the `id` attribute of a `<spf:execute>` action (see section 4.2.3).

- If there is a value specified for the `ref` attribute, only the SPIT test result with an `id` (see section 4.2.3) equal to the `ref` value is evaluated.
- If there is no value specified for the `ref` attribute, all SPIT test results are evaluated.

Depending on the `resultOnMatch` attribute, the `<spf:challenge>` condition behaves as follows:

- If `resultOnMatch` is "true": `<spf:challenge>` evaluates to TRUE if there is any SPIT test result matching

all subconditions; If there is no result matching all subconditions, `<spf:challenge>` evaluates to `FALSE`.

- If `resultOnMatch` is “false”: `<spf:challenge>` evaluates to `FALSE` if there is any SPIT test result matching all subconditions; If there is no result matching all subconditions, `<spf:challenge>` evaluates to `TRUE`.

In other words: if all subconditions of the `<spf:challenge>` match, its result is equal to the value specified in the `resultOnMatch` attribute.

The conditions, which must be met for a SPIT test result to match, are specified using subconditions within the body of the `<spf:challenge>` element. We define the following subconditions: `<spf:eq>`, `<spf:gt>`, `<spf:lt>`, `<spf:geq>`, `<spf:leq>`, `<spf:notSet>`, and `<spf:regEx>`. Of course, the framework can be extended by further subconditions.

Each of the subconditions expects a mandatory `name` attribute specifying the name of a SPIT test result value.

- `eq (neq)`: Evaluates to `TRUE` if the return value of the SPIT test result value specified in the `name` attribute is (not) equal to the specified content.
- `gt (lt)`: Evaluates to `TRUE` if the return value of the SPIT test result value specified in the `name` attribute is greater (less) than the specified content.
- `geq (leq)`: Evaluates to `TRUE` if the return value of the SPIT test result value specified in the `name` attribute is greater (less) than or equal to the specified content.
- `notSet`: Evaluates to `TRUE` if the SPIT test did not return a value with the name specified in the `name` attribute.
- `regEx`: Evaluates to `TRUE` if the return value of the SPIT test result value specified in the `name` attribute matches the regular expression specified in the element’s body.

Policy examples showing the usage of the `<spf:challenge>` condition and its subconditions are presented in section 4.3.

4.2.3 Execute action and action priorities

The `<spf:challenge>` condition defined in section 4.2.2 evaluates the results of previously executed SPIT protection mechanisms. To actually execute these mechanisms, we define the new `<spf:execute>` action.

Our new `<spf:execute>` action, is similar to the `<spit:execute>` action defined in [21]. However, compared to `<spit:execute>` we add two new attributes: `priority` and `id`. A minimal example is shown in listing 4. The `<spf:execute>` action is used to execute SPIT protection mechanisms, such as forwarding a call to a different location, accepting a call, blocking a call, applying call scoring tests [13] or applying a voice CAPTCHA [14].

The new `priority` attribute is optional. This attribute is used to allow for personalized rules which overwrite the default action. Its default value must be specified by the application. Within this paper we assume a default priority of 5. The new `id` attribute is also optional and specifies the name of the variable that is used to store the set of attribute-value pairs returned by the test.

Within the body of the `<spf:execute>` tag, the following values are allowed:

```
<cp:actions>
  <spf:execute priority="2" id="captcha">
    sip:dtmf@test.com
  </spf:execute>
</cp:actions>
```

Listing 4: Execute action example

1. **Block**: The call should be blocked. (Integer value: 1)
2. **URI**: The SPIT protection method specified by the URI should be executed. In general, any URI according to RFC 3986 [3] can be specified. It is up to the application to correctly execute the corresponding SPIT test. The URI could, e.g., specify a SIP URI of a voice CAPTCHA or the SIP address of the user’s voicemail system, in which case the application would have to transfer the call. For a HTTP URI, the application could, e.g., execute a Web Service. (Integer value: 2)
3. **Allow**: The call should be forwarded to its original destination. (Integer value: 3)

According to RFC 4745 [16] the combining rules for each action must be defined. For the `<spf:execute>` action we define the combining rules as follows:

- If two `<spf:execute>` actions are combined, the combined value is equal to the value of the `<spf:execute>` element with the lower value of the `priority` attribute. (The lower the value of the `priority` attribute, the more important the action. E.g., an action with `priority=1` is more important than one with `priority=2`.)
- If the `priority` is equal for both elements the combining rules for integer values, specified in section 10.2 or RFC 4745 [16], apply. The integer values are given in the list above.

For integer [...] permissions, the resulting permission is the maximum value across the permission values in the matching set of rules.

- When two `<spf:execute>` actions are combined, which specify two different URIs and which have the same `priority`, the application should choose the URI leading to the “least restrictive” action (choosing the least restrictive action is one of the general concepts of RFC 4745 [16]). E.g., a DTMF based CAPTCHA is more restrictive than a Voice Energy CAPTCHA, which in turn, is more restrictive than forwarding the call to the user. In order to evaluate this rule, the application must have knowledge about which action is executed when a call is forwarded to a certain URI. In cases where a clear decision cannot be taken, the application shall choose one of the URIs randomly, whereby no URI should be chosen more than once during the processing of a single call. Latter avoids potential loops in the policy processing.

The methods for combining rules are best explained using an example. Table 1 shows the outcome of combining two `<spf:execute>` actions with different values and priorities.

Table 1: Examples for action combination with priorities

Row	Execute 1 value	Execute 1 priority	Execute 2 value	Execute 2 priority	Result
#1	block		block		block
#2	block		allow		allow
#3	block		URI		URI
#4	URI		allow		allow
#5	block	2	allow	2	allow
#6	block	2	allow	7	block
#7	URI	2	allow		URI

- In rows 1 to 4 no priority value is specified, i.e., the default priority of 5 is used.
- In rows 1 to 5 the priority of the first `<spf:execute>` action is equal to the priority of the second action. Thus, the default combining rules for integer values are used. These rules specify that for integer values, the resulting value is the maximum of the input values, e.g., in row 2 the combination of `block` (integer value 1) and `allow` (integer value 3) is `allow`, since $\max(1, 3) = 3$.
- In row 6 the priority of the first action is higher than the priority of the second action. Thus the combined value is equal to the value of the first action, i.e., `block`.

Note that the action priority mechanism defined in this section is orthogonal to the rule level concept described in section 4.2.1. Rule levels are used to prioritize which rules should be evaluated first. Action priorities are used to override the default scheme for combining rules. In combination, both mechanisms allow the consistent evaluation of role-based policies. These mechanisms directly relate to our requirements to support role-based policies, personalization, and resolving policy conflicts (see section 2).

When compared with [21] we define only a single action type (`<spf:execute>`) while [21] defines two different types (`<spit:execute>` and `<spit:forward-to>`). We intentionally limit our framework to a *single* action type because one otherwise risks unresolvable action conflicts. E.g.: With the action definitions as proposed in [21] there are cases where two different types of actions can be active at the same time. In the example in section 6.3 of [21] the two action `<spit:forward-to>sip:answering-machine@...</spit:forward-to>` and `<spit:execute>block</spit:execute>` can occur simultaneously. This means that the application would have to block the call, i.e., reply with a “403 forbidden” message, and forward the call to the answering machine, at the same time. This is clearly impossible and the application would thus have to choose one of the actions. This however is conflicting with the requirements of RFC 4745 [16] which states that *all* actions *must* be executed.

4.2.4 Set transformation

The new `<spf:set>` transformation is used to set additional parameters for the executed actions. Examples for such parameters are the preferred language of the message played by a voice CAPTCHA or the number of allowed retries before such a test is marked as failed.

The mandatory `name` attribute specifies the name of the variable to be set, while the actual body of the transformation specifies the content of the variable. An optional

`priority` argument may be specified, which works similar to the `priority` argument of the `<spf:execute>` action.

According to RFC 4745 [16] the combining rules for each transformation must be defined. For the `<spf:set>` transformation we define the combining rules as follows:

- If two `<spf:set>` transformations are combined, and if the transformations have the same value for the `name` attribute, the combined value is equal to the value of the `<spf:set>` element with the higher priority (specified using the `priority` attribute; the lower the value of the `priority` attribute, the higher the priority).
- Otherwise (i.e., the priority is equal for both elements, or the elements have different `name` attributes) the combining rules for set values, specified in section 10.2 or RFC 4745 [16], apply:

For sets [the resulting permission] is the union of values across the permissions in the matching rule set.

4.3 Policy Examples

Listings 5-7 show examples of rules which could be deployed in an enterprise scenario. Listing 5 shows the rules applied for every user, i.e., company-wide policies. Listings 6 and 7 show example rules for a single user and a “Manager” role, respectively.

Listing 5 contains four different rules. Note that the first three rules contain a `<spf:rule-level>` condition for rule-level 1, i.e., they are amongst the first rules to be evaluated. The first rule (“`spitScore`”) applies a SPIT scoring function if no such function has been applied before. The second rule (“`highScore`”) applies a second SPIT test “`hashCash`” [9] when the SPIT score returned by the first test is larger than 10. The result of this test is evaluated by the third rule (“`hashCashFailed`”). If the test was failed, the call is blocked. The last rule of listing 5 defines a default behavior. As it has no conditions defined, except for the a rule level of 10, this rule always belongs to the matching rule set for level 10. Thus, if no other action was found until the 10th rule level is reached, the call will be forwarded to the user.

The rules shown in listing 6 are additional rules for a single user. As all these rules contain a `<spf:rule-level>` condition for level 2 they are only evaluated when the company-wide rules of listing 5 did not result in any action. The user configured two additional rules which, depending on the SPIT score, either forward the call to his voicebox or block the call directly. Note that for a total SPIT score greater than 20 both rules, “`voiceMail`” and “`blockVeryHighScore`”, belong to the matching rule set. Assume for a moment the case that no `priority` attribute was specified for the

```

<cp:rule id="spitScore">
  <cp:conditions>
    <spf:rule-level>1</spf:rule-level>
    <spit:spit-handling>
      <spf:challenge resultOnMatch="false">
        <spf:eq name="method">spitScore</spf:eq>
      </spf:challenge>
    </spit:spit-handling>
  </cp:conditions>
  <cp:actions>
    <spf:execute>http://spitScore</spf:execute>
  </cp:actions>
</cp:rule>

<cp:rule id="highScore">
  <cp:conditions>
    <spf:rule-level>1</spf:rule-level>
    <spit:spit-handling>
      <spf:challenge>
        <spf:eq name="method">spitScore</spf:eq>
        <spf:gt name="total-score">10</spf:gt>
      </spf:challenge>
    </spit:spit-handling>
  </cp:conditions>
  <cp:actions>
    <spf:execute
      id="hashCash">sip:hashCash</spf:execute>
  </cp:actions>
</cp:rule>

<cp:rule id="hashCashFailed">
  <cp:conditions>
    <spf:rule-level>1</spf:rule-level>
    <spit:spit-handling>
      <spf:challenge ref="hashCash">
        <spf:eq name="result">failed</spf:eq>
      </spf:challenge>
    </spit:spit-handling>
  </cp:conditions>
  <cp:actions>
    <spf:execute priority="1">block</spf:execute>
  </cp:actions>
</cp:rule>

<cp:rule id="defaultAllow">
  <cp:conditions>
    <spf:rule-level>10</spf:rule-level>
  </cp:conditions>
  <cp:actions>
    <spf:execute>allow</spf:execute>
  </cp:actions>
</cp:rule>

```

Listing 5: Example – company wide rules

```

<cp:rule id="voiceMail">
  <cp:conditions>
    <spf:rule-level>2</spf:rule-level>
    <spit:spit-handling>
      <spf:challenge>
        <spf:eq name="method">spitScore</spf:eq>
        <spf:gt name="total-score">10</spf:gt>
      </spf:challenge>
    </spit:spit-handling>
  </cp:conditions>
  <cp:actions>
    <spf:execute>
      sip:voicemail@company
    </spf:execute>
  </cp:actions>
</cp:rule>

<cp:rule id="blockVeryHighScore">
  <cp:conditions>
    <spf:rule-level>2</spf:rule-level>
    <spit:spit-handling>
      <spf:challenge>
        <spf:eq name="method">spitScore</spf:eq>
        <spf:gt name="total-score">20</spf:gt>
      </spf:challenge>
    </spit:spit-handling>
  </cp:conditions>
  <cp:actions>
    <spf:execute priority="1">block</spf:execute>
  </cp:actions>
</cp:rule>

```

Listing 6: Example – Single user rules

“**blockVeryHighScore**” rule. In this case, according to the the action merging rules defined in section 4.2.3, the resulting action would always be the forwarding to the voicemail. However, as there *is* a **priority** attribute specified, the **block** action wins in the case of a total SPIT score larger than 20.

The last rule example is listed in listing 7. This rule is applied to all users in the *Manager* role. It makes sure that calls are forwarded to the voicemail system if the called person is currently in a meeting. The used **<spit:presence-status>** is defined in [21]. Note that, due to the rule-level of 3, this rule is only evaluated when none of the SPIT related rules matches.

```

<cp:rule id="inMeeting">
  <cp:conditions>
    <spf:rule-level>3</spf:rule-level>
    <spit:presence-status>
      meeting
    </spit:presence-status>
  </cp:conditions>
  <cp:actions>
    <spf:execute>
      sip:voicemail@company
    </spf:execute>
  </cp:actions>
</cp:rule>

```

Listing 7: Example – Manager rule

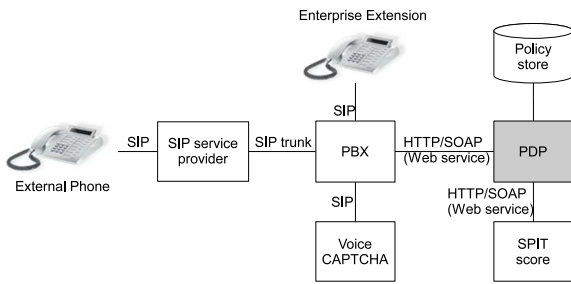


Figure 2: Architecture of Prototypical Implementation

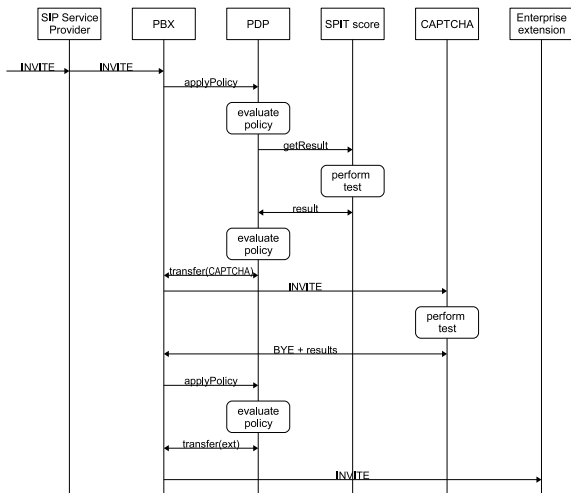


Figure 3: Message Flow in Prototype (simplified)

5. PROTOTYPE IMPLEMENTATION

We implemented a prototypical PDP (Policy Decision Point) for the proposed framework and integrated it with a NEC SIP PBX. The integration was performed by using the web service interfaces offered by this COTS (Commercial off-the-shelf) software. For the SPIT protection tests we use the VoIP SEAL framework [6] which includes, amongst others, a SPIT scoring function as well as different voice CAPTCHAs. Figure 2 shows the functional entities of our prototypical implementation and the communication interfaces between them.

In our implementation the PDP controls the PBX using the web service interface provided by the PBX. The SPIT scoring test is also implemented as a web service and is directly queried by the PDP when needed. The voice CAPTCHA is realized as a separate SIP User Agent (UA) which registers itself with the PBX.

A simplified message chart is depicted in figure 3. For simplicity, many messages, such as SIP reply messages or messages involved in the session transfers, are not shown.

When a new external call arrives at the PBX, the PBX does not immediately forward the call to its intended destination, but first queries the PDP to apply the SPIT policies. The call is put on hold in the meantime. The PDP checks the policy store for applicable rules, evaluates them and computes the merged action for the matching rule set. The example shown in figure 3 assumes that, as a result of the policy evaluation, the first action is to determine a SPIT

score for the incoming call. Thus the PDP has to query the SPIT score web service. After the calculated SPIT score is returned to the PDP, the policies are re-evaluated, now also taking into account the new input variables of the SPIT score. We assume the outcome of the second policy evaluation is to apply a voice CAPTCHA. In this case, the call must be transferred to the UA of the CAPTCHA. The transfer is initiated using a web service request from the PDP to the PBX. The PBX takes care of the signaling for the actual call transfer. The CAPTCHA answers the call, performs the test and returns the test results back to the PDP. The results can, e.g., be included in the BYE message sent by the CAPTCHA, or can be stored in a data store shared by the PDP and the CAPTCHA. Assuming the CAPTCHA was passed, the policies are evaluated a third time and the call is – in this example – forwarded to the intended receiver, i.e., the enterprise extension.

For managing policies and monitoring the system we developed a Web GUI. As an example, figure 4 shows the call history screen. This screen contains the list of calls previously processed by the PDP and the corresponding policy evaluation results. For the second call in figure 4 three different actions have been executed. First, a SPIT scoring function was applied (within the VoIP SEAL framework we used [6] “stage1” represents a SPIT scoring function). Afterwards, the user was redirected to a voice CAPTCHA (“turingTest”) located at the extension number 8912. Finally the call was forwarded to the intended receiver (“allow”). Furthermore, a transformation with the value “language=de” was active, i.e., the CAPTCHA was played in German language.

6. CONCLUSION

Spam over IP Telephony (SPIT) is a serious threat for VoIP service architectures. At the same time, SPIT is a very personal and subjective matter because different users may have different preferences with respect to unsolicited communications. Since no solution can offer *perfect* SPIT protection (i.e., a 0-percent false positive rate and a 0-percent false negative rate), a personalizable and parametrizable SPIT protection solution is necessary.

In this paper, we present the design and implementation of a policy framework which enables parametrizable, personalized, and role-based SPIT prevention. Our approach allows the customized application of different SPIT protection mechanisms without making any assumptions on the type of SPIT test applied to incoming messages nor on the type of test results. The proposed framework is based on Common Policy (RFC 4745 [16]) and reuses previous (but insufficient) approaches of Common Policy extensions for SPIT prevention [21] as much as possible. By introducing two new concepts, *rule-levels* and *action priorities*, we

Call History															
From	To	Status	Last Action												
0123	12	disconnected	[ExecuteAction [null] block]												
0123	12	disconnected	<table border="1"> <thead> <tr> <th>Iteration</th> <th>Action</th> <th>Transformations</th> </tr> </thead> <tbody> <tr> <td>1</td> <td>[ExecuteAction [stage1] stage1]</td> <td>{language={de}}</td> </tr> <tr> <td>2</td> <td>[ExecuteAction [turingTest] 8912]</td> <td>{language={de}}</td> </tr> <tr> <td>3</td> <td>[ExecuteAction [null] allow]</td> <td>{language={de}}</td> </tr> </tbody> </table>	Iteration	Action	Transformations	1	[ExecuteAction [stage1] stage1]	{language={de}}	2	[ExecuteAction [turingTest] 8912]	{language={de}}	3	[ExecuteAction [null] allow]	{language={de}}
Iteration	Action	Transformations													
1	[ExecuteAction [stage1] stage1]	{language={de}}													
2	[ExecuteAction [turingTest] 8912]	{language={de}}													
3	[ExecuteAction [null] allow]	{language={de}}													

Figure 4: Call history screen of the Web GUI

enable the combination of multiple policies in a role-based policy scenario. Further, we add extensions which provide means to execute personalized SPIT protection mechanisms as well as to make fine granular decisions, depending on the results of individual SPIT tests.

As a proof of concept, we implemented our proposed framework in a prototypical Policy Decision Point (PDP) and integrated it with a commercial off-the-shelf SIP PBX. We have implemented several use cases in this prototype, demonstrating the feasibility of our approach. During the process of integration and implementing certain scenarios in practice, we learned the shortcomings of previous SPIT policy extensions and derived the necessity for our new concepts.

As future work in this area we consider the prototypical implementation of a sophisticated graphical user interface. Such an interface would allow administrators and users to specify SPIT protection preferences for our framework in a user-friendly way. In addition, we consider contributing our findings to the IETF, most probably in cooperation with the authors of [21] which we already contacted.

7. REFERENCES

- [1] Cisco Security Advisory: Multiple Product Vulnerabilities Found by PROTON SIP Test Suite. available online at <http://www.cisco.com/warp/public/707/cisco-sa-20030221-protos.shtml>.
- [2] V. A. Balasubramanian, M. Ahamad, and H. Park. CallRank: Combating SPIT Using Call Duration, Social Networks and Global Reputation. In *CEAS 2007 Fourth Conference on Email and AntiSpam*, 2007.
- [3] T. Berners-Lee, R. Fielding, and L. Masinter. Uniform Resource Identifier (URI): Generic Syntax. RFC 3986 (Standard), Jan. 2005.
- [4] L. Blair and K. J. Turner. Handling Policy Conflicts in Call Control. In *Proc. 8th International Conference on Feature Interaction*, pages 39–57. IOS Press, Amsterdam, June 2005.
- [5] N. Damianou, A. K. Bandara, M. Sloman, and E. C. Lupu. A Survey of Policy Specification Approaches. available online at <http://en.scientificcommons.org/552923>.
- [6] N. d’Heureuse, J. Seedorf, S. Niccolini, and T. Ewald. Protecting SIP-Based Networks and Services from Unwanted Communications. In *Proc. IEEE Global Telecommunications Conference IEEE GLOBECOM 2008*, pages 1–5, 2008.
- [7] S. Duflos, G. Diaz, V. Gay, and E. Horlait. A Comparative Study of Policy Specification Languages for Secure Distributed Applications. In *Management Technologies for E-Commerce and E-Business Applications, LNCS 2506*, 2002.
- [8] D. Ferraiolo and R. Kuhn. Role-based access controls. In *In 15th NIST-NCSC National Computer Security Conference*, pages 554–563, 1992.
- [9] C. Jennings. Computational Puzzles for SPAM Reduction in SIP, draft-jennings-sip-hashcash (expired). Internet Engineering Task Force, July 2007.
- [10] D. Jiang, R. Liscano, and L. Logrippo. Personalization of Internet Telephony Services for Presence with SIP and Extended CPL. In *Computer Communications*, 29 (18), pages 3766–3779, November 2006.
- [11] J. Lennox, X. Wu, and H. Schulzrinne. Call Processing Language (CPL): A Language for User Control of Internet Telephony Services. RFC 3880 (Proposed Standard), Oct. 2004.
- [12] S. Niccolini, K. Fischer, D. Wing, M. Stiernerling, and H. Tschofenig. Spam feedback for SIP, draft-niccolini-sipping-spam-feedback (expired). Internet Engineering Task Force, Feb. 2008.
- [13] J. Quittek, S. Niccolini, S. Tartarelli, and R. Schlegel. On Spam over Internet Telephony (SPIT) Prevention. *IEEE Communications Magazine*, 46(8):80–86, 2008.
- [14] J. Quittek, S. Niccolini, S. Tartarelli, M. Stiernerling, M. Brunner, and T. Ewald. Detecting SPIT Calls by Checking Human Communication Patterns. In *Proc. IEEE International Conference on Communications ICC ’07*, pages 1979–1984, 24–28 June 2007.
- [15] J. Rosenberg, G. Camarillo, and D. Willis. A Framework for Consent-Based Communications in the Session Initiation Protocol (SIP). RFC 5360 (Proposed Standard), Oct. 2008.
- [16] H. Schulzrinne, H. Tschofenig, J. Morris, J. Cuellar, J. Polk, and J. Rosenberg. Common Policy: A Document Format for Expressing Privacy Preferences. RFC 4745 (Proposed Standard), Feb. 2007.
- [17] C. Sorge and J. Seedorf. A Provider-Level Reputation System for Assessing the Quality of SPIT Mitigation Algorithms. In *Proceedings of IEEE ICC 2009 (to appear)*, 2009.
- [18] Y. Soupionis, S. Dritsas, and D. Gritzalis. An Adaptive Policy-Based Approach to SPIT Management. In *ESORICS ’08: Proceedings of the 13th European Symposium on Research in Computer Security*, pages 446–460, Berlin, Heidelberg, 2008. Springer-Verlag.
- [19] H. Tschofenig, G. Dawirs, T. Froment, D. Wing, and H. Schulzrinne. Requirements for Authorization Policies to tackle Spam and Unwanted Communication for Internet Telephony, draft-froment-sipping-spit-requirements (expired). Internet Engineering Task Force, July 2008.
- [20] H. Tschofenig, E. Leppanen, S. Niccolini, and M. Arumaithurai. Completely Automated Public Turing Test to Tell Computers and Humans Apart (CAPTCHA) based Robot Challenges for SIP, draft-tschofenig-sipping-captcha (expired). Internet Engineering Task Force, Feb. 2008.
- [21] H. Tschofenig, D. Wing, H. Schulzrinne, T. Froment, and G. Dawirs. A Document Format for Expressing Authorization Policies to tackle Spam and Unwanted Communication for Internet Telephony, draft-tschofenig-sipping-spit-policy (expired). Internet Engineering Task Force, July 2008.
- [22] A. Westerinen, J. Schnizlein, J. Strassner, M. Scherling, B. Quinn, S. Herzog, A. Huynh, M. Carlson, J. Perry, and S. Waldbusser. Terminology for Policy-Based Management. RFC 3198 (Informational), Nov. 2001.
- [23] D. Wing, S. Niccolini, M. Stiernerling, and H. Tschofenig. Spam Score for SIP, draft-wing-sipping-spam-score (expired). Internet Engineering Task Force, Feb. 2008.