Gray Policy Specification

Gabriel Laverghetta

***Abstract*—Policy specification is a vital task in the realm of software security. A wide variety of policy specification languages have been developed, but none support the gray model of security policies. This paper describes changes that were made to the PoCo policy specification language. These changes have given the language the ability to define gray security policies. Two examples of gray policies defined with the language are presented. The paper also discusses some advantages and disadvantages of gray policies and how they may be useful in practice.**

Keywords—policies, gray policies, policy specification languages

I. Introduction

Security policies are a foundational concept in software security. Software is said to be secure when it adheres to constraints laid out by policies. Numerous methodologies have been developed to specify policies. The set-based definition, for example, considers policies to be sets of "good" program traces; only those traces in the set are permitted to execute. Similarly, policies may be specified using predicates on sets of executions [1]. While these definitions are useful, policies specified with them are difficult to translate into software. The task of creating policies for software has been made easier using policy specification languages. These programming languages aid policymakers in the specification and enforcement of security policies [2]. Many Turing-complete policy specification languages have been developed for a variety of domains [3, 4]. The goal of this paper is to present a version of the PoCo (Policy Composition) policy specification language that has been modified to support the gray model of security policies.

Review of related works. This paper sits at the intersection of the theory of gray policies and the PoCo programming language.

Gray security policies form an extension of traditional, black and white policies [5]. Rather than simply stating if programs or program executions are secure, gray policies assign a security rating to program executions. These ratings are real numbers in some interval, (for example, [0,1]) where the minimum value indicates complete insecurity and the maximum indicates complete security. In the gray model, program traces are sequences of exchanges. An exchange is a tuple <e, e'> of program events, where e is the event that the target system intends to execute and e' is the event that is executed. The set of finite and infinite program traces is defined as ε^{∞}. The gray model's definitions of black and white policies (1) and gray policies (2) are as follows:

$$2^{\varepsilon^{\infty}} \rightarrow \{false, true\} \tag{1}$$

$$2^{\varepsilon^{\infty}} \rightarrow \mathbb{R}_{[0,1]} \tag{2}$$

Many security policies restrict which program actions are permitted. Mechanisms that enforce these policies may simply terminate the execution if the policy is violated [1]. More complicated policies may specify obligations. An obligation alters the flow of a program through the insertion of output events into traces in response to certain inputs [6]. The PoCo policy specification language allows programmers to write complex policies, including policies that contain obligations. The primary goal of the language is to handle conflicts between policies and allow policies to react to obligations.

Contributions and roadmap. This paper presents a fork of PoCo that has support for gray policies. Section 2 discusses the capabilities that the modified language should possess. Section 3 presents the actual changes that have been made to PoCo's code. Section 4 describes the environment used to develop test the changes. In Section 5, the gray policies that have been constructed using the modified language are illustrated. Some of the advantages and drawbacks of the language are discussed in Section 6. Finally, Section 7 concludes and explores directions for future work.

II. Goals of the Modified Language

Before delving into the implementation details, it is worth considering what capabilities a gray policy specification language should possess. One of the primary advantages of the gray model is that it allows users to compare the relative security of systems [5]. This allows users to decide how much security they desire, as opposed to choosing between complete security and complete insecurity. Therefore, the modified language should assign security values or ratings to program executions. These ratings measure how well the policies are being followed. For example, a program might begin with a security rating of 100 (the maximum possible rating), and insecure actions would decrease this rating (the minimum rating being 0).

Although we wish to use these quantitative ratings, there are still binary decisions to be made. Consider the use case of an email client (an email client was also used to test the original PoCo language [6]). The client might adhere to a security policy requiring that emails from unknown addresses (or addresses not found in a list of known addresses) be marked as spam. Marking emails as spam is a binary decision; either the emails are marked or they are not. Constructing a gray policy for this requirement could be done in various ways. One approach might be to keep track of the number of emails received from each unfamiliar sender. There could be a security rating associated with each sender; each email received from them would decrease their security rating. As the number of spam emails increases, so does the severity of the policy violation. To inform the user of the policy violation, a message could be

Table 1: Spam email policy.

Security rating interval	Phrase prepended to email subject line
[100, 100]	[none]
[90, 100)	POSSIBLE SPAM:
[70, 90)	LIKELY SPAM:
[50, 70)	WARNING! LIKELY SPAM:
[0, 50)	DANGER! VERY LIKELY SPAM:

placed before the subject line of incoming emails. The black and white version of this policy simply prepends “SPAM?” to emails from unfamiliar senders. To implement the gray policy, one could prepend the security rating to each incoming email, but this would not be user-friendly. Another approach would involve assigning phrases to intervals. Table 1 shows the phrases that could be placed before the subject lines of emails from unknown senders. As more and more possible spam is received, the security rating of the sender is lowered. The warnings become more and more dire at a commensurate rate.

Of course, one could argue that this gray policy is merely a conjunction of black and white policies. One could construct black and white policies mandating that messages like those from Table 1 be prepended to emails depending on the number of emails received. Although there is truth to this, with the gray model there is just one policy that implements this functionality. Managing one policy is simpler and less error-prone than attempting to compose numerous policies. With the gray policy in place, there are security values associated with possible senders of spam, allowing administrators and policymakers direct control over the appearance of emails from these senders. This also gives administrators a real-time view of which addresses are generating the most spam, which might inform security-related decisions such as blocking those addresses.

One implementation detail involves how a sender’s security rating should be adjusted when possible spam is received. Policymakers could choose from any number of mathematical functions to adjust the security rating. To illustrate, let R(x) determine the sender’s security rating, M be the maximum possible rating, and x be the number of spam emails received. The ratings could be adjusted by a linear function (3), decimation (4), or a quadratic function (5), as shown below:

$$R(x) = M - cx \tag{3}$$

$$R(x) = \begin{cases} 100, & x = 0 \\ 0.9R(x-1), & x > 0 \end{cases} \tag{4}$$

$$R(x) = M - x^2 \tag{5}$$

The rating could simply be decreased by a constant value (say, ten) for each email. A more complicated approach would be to decimate the security rating after each email (this decimation approach was discussed in [5]). With this approach, the security rating decreases quickly at first but changes less and less as more emails are received. Using the quadratic function M – x^2 would decrease the rating slowly in response to the first few spam emails, but receiving many more such emails would drastically lower the rating. This is likely the behavior that most would desire. Both functions are graphed in Fig. 1.

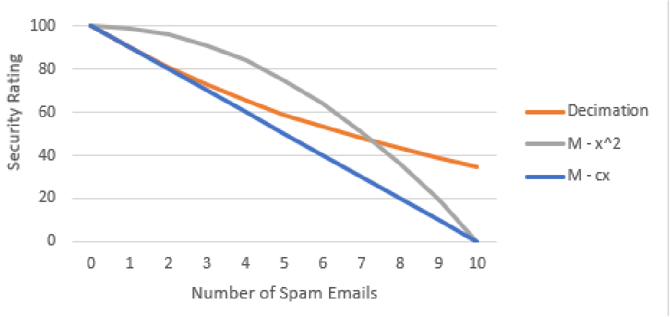


Fig. 1. Graph of the security rating functions. M = 100, c = 10.

To summarize, gray policy specification languages should maintain security ratings that judge how closely policies are being followed. These ratings inform how the target system’s output should be adjusted; more severe violations may require more significant adjustments. The ratings are governed by mathematical functions that precisely measure deviation from policies.

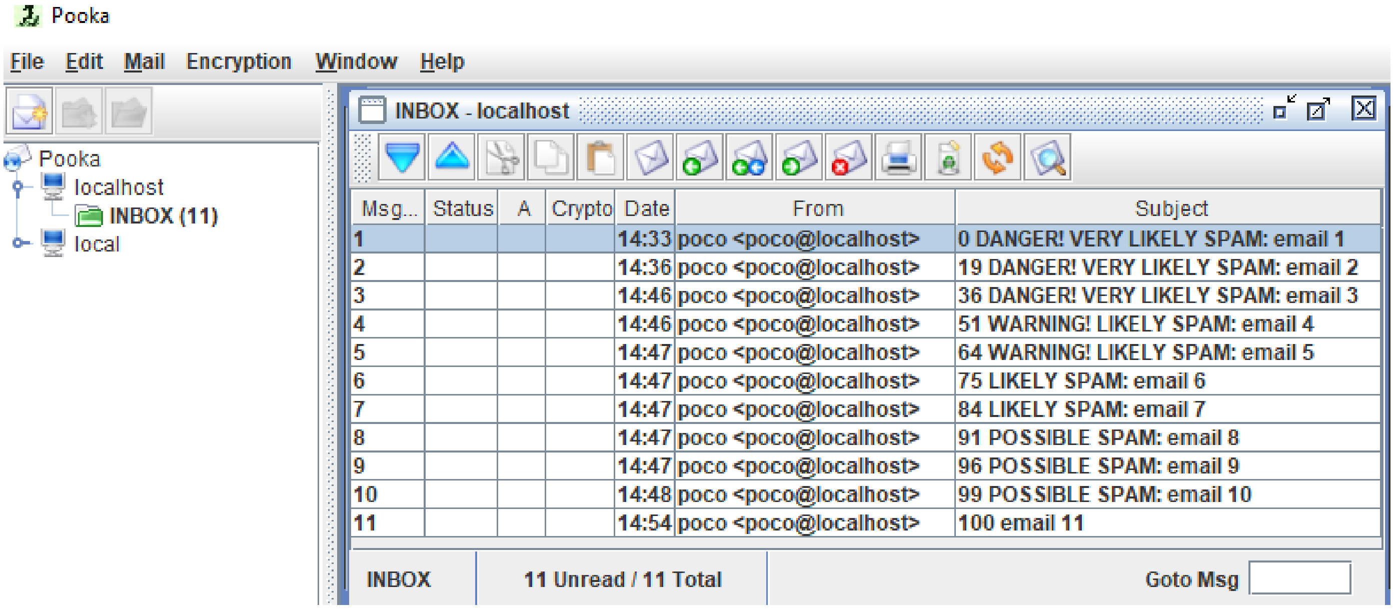
As a final requirement, none of the changes proposed in this paper should interfere with the stated goals of the original PoCo language. PoCo is the first language to support atomic obligations, conflict resolution between obligations, and the ability of policies to react to obligations [6]. Ideally, the modified version of PoCo would continue to fulfill these objectives.

III. PoCo Modifications

PoCo is a Java application developed with the Eclipse IDE. It uses the AspectJ extension to inline security policies into target applications. The PoCo technical manual [7] contains a full description of the language, and the PoCo source code is available on GitHub [8]. This paper’s fork of the language is also available online [9].

PoCo defines a Policy class to represent security policies. The fork defines a GrayPolicy class that extends the Policy class. A GrayPolicy has a private field securityValue, which is the value/rating indicating how well the policy is being followed. The GrayPolicy class contains an accessor and mutator method for securityValue. This class also contains two static constants, one for the maximum possible securityValue and one for its minimum. Since these constants are static, they are the same for all instances of GrayPolicy. As a result, the security values of all GrayPolicies have the same maximum and minimum. This is a design choice that helps maintain consistency among policies.

Several changes needed to be made to PoCo’s code to support the new GrayPolicy class. Before PoCo begins



Msg...	Status	A	Crypto	Date	From	Subject
1				14:33	poco <poco@localhost>	0 DANGER! VERY LIKELY SPAM: email 1
2				14:36	poco <poco@localhost>	19 DANGER! VERY LIKELY SPAM: email 2
3				14:46	poco <poco@localhost>	36 DANGER! VERY LIKELY SPAM: email 3
4				14:46	poco <poco@localhost>	51 WARNING! LIKELY SPAM: email 4
5				14:47	poco <poco@localhost>	64 WARNING! LIKELY SPAM: email 5
6				14:47	poco <poco@localhost>	75 LIKELY SPAM: email 6
7				14:47	poco <poco@localhost>	84 LIKELY SPAM: email 7
8				14:47	poco <poco@localhost>	91 POSSIBLE SPAM: email 8
9				14:47	poco <poco@localhost>	96 POSSIBLE SPAM: email 9
10				14:48	poco <poco@localhost>	99 POSSIBLE SPAM: email 10
11				14:54	poco <poco@localhost>	100 email 11

Fig. 2. Screenshot of Pooka showing 11 emails from the same unknown source.

applying policies to the target application, it must compile the policies specified and perform static analysis on them. During this process there are references to the Policy class in several places in the code. For example, the visitClass method in PolicyScanner.java will only visit instances of the Policy class. To support gray policies, these points in the code were updated to also check for instances of the GrayPolicy class. A full list of these changes is present in the README file in [9].

Aside from general language changes, this paper introduces two new gray policies to PoCo. These are both "gray versions" of black and white policies defined in the original language. The first policy, defined in SpamPolicy.java, implements the spam email behavior discussed in Section 2. The second policy, defined in the same file, deals with email attachments. Both policies are discussed in Section 5.

IV. Development and Testing Environment

As a Java application, Eclipse is the IDE of choice for developing PoCo. The environment used to test the new policies and language changes is the same environment used to test the original PoCo language [6] and the Polymer language [10]. The two policies defined in Section 5 were applied to the Pooka [11] open-source email client. Emails were stored locally on a GreenMail [12] email server. Using local email servers for development and testing is a common practice with numerous benefits [13].

V. Examples of Newly defined Policies

With the modifications introduced in Section 3 in place, two gray policies have been implemented.

Spam email policy. The spam email policy works as described in Section 2. The number of emails sent by each unknown address is stored in a HashMap. As the same unknown source sends more and more emails, the subject lines of these emails are prepended with more and more urgent warnings. Figure 2 shows the Pooka client after the user has received 11 emails from an unknown sender. The sender's security value has been prepended to the subject lines in addition to the warning. In a deployed application, this number would probably not be shown to users, but it is useful to show for testing purposes.

Email attachment policy. Email attachments are serious security vulnerabilities [14]. PoCo defines a black and white policy that displays a pop-up when users attempt to open an email from an unknown sender that contains a potentially dangerous attachment. Potentially dangerous attachments are defined as those that are not plain text files. The gray version of this policy takes a closer look at the attachment's type. Among the file types that are not plain text, some are more dangerous than others. The gray policy assigns security ratings to emails containing attachments based on the attachment's file type.

Table 2 shows the security ratings of file types along with explanations for the rating. This is not an exhaustive list of all file formats, but it does contain representative examples from five classes of files. The safest class of files are those that contain only plain text. Compressed file archives are questionable – they may contain dangerous files, but this is not known until they are downloaded and extracted. Source code files are dangerous, but they must be compiled using tools that end-users may not have at their disposal. The fourth class, executable files, is quite dangerous because these files can be run easily. This policy considers the most dangerous file type to be macro-enabled Microsoft Office files. Office macros have nearly the same capabilities as other forms of malware [15], and as of August 2018 they were one of the most common malware delivery mechanisms [16]. In addition, users often expect to see Office files as attachments in emails. These files are transmitted so commonly that users may not even notice the "m" in the docm file extension. Users might "let their guard down" if an Office file is attached, but this is less likely to be the case with the other file classes.

Table 2: Email Attachment Policy.

Security rating	File type	Explanation
100	txt	safe – plain text
75	zip	questionable – compressed archive may contain dangerous files
50	c	dangerous – source code may be malicious, but must be compiled to run
25	exe	very dangerous – executable code has been compiled and may be run easily
0	docm	extremely dangerous – users likely to trust this since it is a Word document, but it contains embedded code (macros)

Like the black and white email attachment policy, this policy displays a pop-up window when the user attempts to open an email from an unknown address that contains an attachment. A warning message is displayed in the pop-up, and this message changes depending on the type of file attached to the email. As with the spam policy, more dangerous file types will result in more urgent messages appearing in the pop-up. A HashMap is defined that stores mappings between file types and the warning text to display for each type. Figure 3 shows the warning message that is displayed when the user attempts to open an email containing a docm attachment.

VI. Discussion

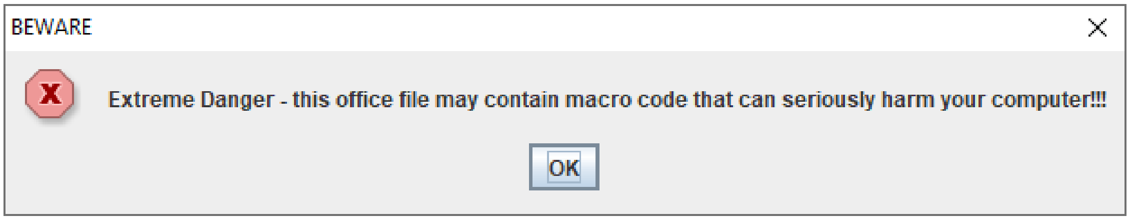
Using gray policies offers policymakers a great deal of control over targets, allowing them to finely tune responses to policy violations. The security ratings measure exactly how much the policy is violated, and the security functions allow these ratings to be adjusted however policymakers see fit. By comparing the security values of policies currently being enforced, administrators can determine which policies may be too lenient or too harsh. A policy with a particularly low value might indicate the need for additional end-user training or a potential attack in progress.

The two policies presented in this paper represent a first foray into creating gray policies for real-world applications. The focus of this paper has been developing the necessary language constructs and implementing the policies themselves. However, there are usability factors that may degrade the effectiveness of gray policies in practice. In particular, the exact words and language used in the spam policy security warnings may greatly affect how users respond to the warnings. Security warnings are an important topic in the field of usable security. Technical jargon and the overuse of the warnings have weakened the impact these warnings have on users [17, 18]. This paper leaves open the questions of how to phrase these warnings and how often to use them.

VII. Conclusions and Future Work

This paper has united the gray theory of security policies and the PoCo specification language. The goals of the PoCo modifications were clarified, and the modifications themselves were presented. Two policies have been created using the language, showing how varying infractions on these policies can lead to varying feedback to the user. The gray model offers numerous benefits to users and security administrators, but it is not without its drawbacks.

There are numerous directions for future work. For starters, not all the ideas explored in the gray model have been addressed in this paper. It would be interesting to see how silhouettes and silhouette judges [5] could be implemented in practice. In addition, the original PoCo paper defined ten policies in its case study [6], but this paper has considered only the spam policy and the email attachment policy. How the remaining policies should be grayed is an open question. Lastly, the gray model is an area that would benefit from usability research. Case studies could be conducted to determine effective ways to communicate policy violations to end-users.



Extreme Danger - this office file may contain macro code that can seriously harm your computer!!!

Fig. 3. Warning message displayed when attempting to open an email containing a macro-enabled office file.

References

[1] F. B. Schneider, "Enforcable Security Policies," in ACM Transactions on Information and Systems Security, Vol. 3, No. 1, pp 30-50, February 2000.

[2] J. Ligatti, B. Rickey, and N. Saigal, "LoPSiL: A Location-based Policy-specification Language," in Security and Privacy in Mobile Information and Communication Systems, pp. 265-277, June 2009.

[3] L. Bauer, J. Ligatti., and D. Walker, "Composing Security Policies with Polymer," in ACM Conference on Programming Language Design and Implementation, 2005.

[4] eXtensible Access Control Markup Language (XACML) version 2.0, http://docs.oasis-open.org/xacml/2.0/access_control-xacml-2.0-core-spec-os.pdf

[5] D. Ray and J. Ligatti, "A Theory of Gray Security Policies," in European Symposium on Research in Computer Security, pp. 481-499, 2015.

[6] D. Ferguson, Y. Albright, D. Lomsak, T. Hanks, K. Orr, and J. Ligatti, "PoCo: A Language for Specifying Obligation-Based Policy Compositions," in Proceedings of the *9th* International Conference on Software and Computer Applications (ICSCA), February 2020.

[7] D. Ferguson, Y. Albright, D. Lomsak, T. Hanks, K. Orr, and J. Ligatti, "Composition of atomic obligation security policies." http://www.cse.usf.edu/ ligatti/papers/PoCoTR.pdf, 2019.

[8] Y. Albright, "PoCo source code." https://github.com/caoyan66/PoCo-PolicyComposition/, 2018.

[9] G. Laverghetta, "Gray Policy Specification", https://github.com/glaverghetta/policyComposition, 2023.

[10] L. Bauer, J. Ligatti, and D. Walker, "Composing expressive runtime security policies," ACM Transactions on Software Engineering and Methodology, vol. 18, no. 3, pp. 1–43, 2009.

[11] A. Petersen, "Pooka: A java email client, 2003." http://www.suberic.net/pooka/.

[12] GreenMail, "Open Source suite of lightweight and sand-boxed email servers supporting SMTP, POP3 and IMAP", https://greenmail-mail-test.github.io/greenmail/.

[13] Mailosaur, "Setting up a fake SMTP server for testing", https://mailosaur.com/blog/setting-up-a-fake-smtp-server-for-testing/.

[14] Y. Miao, "Five Steps to Keep Your Email Attachments Secure," OPSWAT, https://www.opswat.com/blog/five-steps-keep-your-email-attachments-secure, 2014.

[15] "Macro Security for Microsoft Office (2019 Update)," National Cyber Security Centre, https://www.ncsc.gov.uk/guidance/macro-security-for-microsoft-office, 2019.

[16] "Microsoft Office Macros: Still Your Leader in Malware Delivery,"

Cofense Intelligence, https://cofense.com/blog/microsoft-office-macros-still-leader-malware-delivery/, 2018

[17] F. N. A. Ahmad, Z. F. Zaaba, M. A. I. M. Aminuddin, and N. L. Abdullah, "Empirical Investigations on Usability of Security Warning Dialogs: End Users Experience," International Conference on Advances in Cyber Security, pp 335-349, 2019.

[18] A. Amran, Z. F. Zaaba, M. M. Singh, A. W. Marashdih, "Usable Security: Revealing End-Users Comprehensions on Security Warnings," Information Systems International Conference, vol. 124, pp. 624-631, 2017.