Automated Implementation of Windows-related Security-Configuration Guides

Patrick Stöckle
patrick.stoeckle@tum.de
Technical University of Munich
Munich, Germany

Bernd Grobauer
bernd.grobauer@siemens.com
Siemens AG
Munich, Germany

Alexander Pretschner
alexander.pretschner@tum.de
Technical University of Munich
Munich, Germany

ABSTRACT

Hardening is the process of configuring IT systems to ensure the security of the systems' components and data they process or store. The complexity of contemporary IT infrastructures, however, renders manual security hardening and maintenance a daunting task.

In many organizations, security-configuration guides expressed in the *SCAP* (Security Content Automation Protocol) are used as a basis for hardening, but these guides by themselves provide no means for automatically implementing the required configurations.

In this paper, we propose an approach to automatically extract the relevant information from publicly available security-configuration guides for Windows operating systems using natural language processing. In a second step, the extracted information is verified using the information of available settings stored in the *Windows Administrative Template* files, in which the majority of Windows configuration settings is defined.

We show that our implementation of this approach can extract and implement 83% of the rules without any manual effort and 96% with minimal manual effort. Furthermore, we conduct a study with 12 state-of-the-art guides consisting of 2014 rules with automatic checks and show that our tooling can implement at least 97% of them correctly. We have thus significantly reduced the effort of securing systems based on existing security-configuration guides.

CCS CONCEPTS

• **Security and privacy → Software security engineering**; **Usability in security and privacy**; • **Software and its engineering → Software configuration management and version control systems**.

ACM Reference Format:
Patrick Stöckle, Bernd Grobauer, and Alexander Pretschner. 2020. Automated Implementation of Windows-related Security-Configuration Guides. In *35th IEEE/ACM International Conference on Automated Software Engineering (ASE '20), September 21–25, 2020, Virtual Event, Australia.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3324884.3416540

ASE '20, September 21–25, 2020, Virtual Event, Australia

ACM ISBN 978-1-4503-6768-4/20/09…$15.00
https://doi.org/10.1145/3324884.3416540

1 INTRODUCTION

Misconfigurations reduce the security of a system by introducing vulnerabilities that are often difficult to trace. A recent study [6] has demonstrated that from the perspective of the operators there is one major factor for security misconfigurations: lack of knowledge.

One attempt to deal with the lack of knowledge is to use existing security-configuration guides. These guides consist of several rules for a specific software system such as Windows 10 or Red Hat Enterprise Linux. Each rule explains which setting should be set to which value to make the system more secure and why we should apply it (e.g., Listing 1). Known publishers of such guides are the Center for Internet Security (CIS) or the Defense Information Systems Agency (DISA). Organizations and companies like Siemens can use these guides to harden their systems.

One may be tempted to argue that we do not need security configuration because companies like Microsoft make a strong effort to configure their systems securely by default. These companies invest a lot in security, of course, but security is just one concern, in addition to others, including usability. Assume that there was a handy setting for most customers, but it poses a small security risk. The companies may be tempted to decide to have it activated by default, whereas security-aware customers would deactivate it. Similarly, we could argue for the data collection settings. They bring knowledge to the companies to improve their products, and all customers can profit from this. Thus, the companies may be tempted to activate data collection settings by default. In contrast, customers with high-security requirements would deactivate them to reduce the risk that sensitive information is accidentally leaked via the data collection. Thus, security-configuration guides from independent organizations can help security-concerned customers in making their systems more secure.

```
## /rule
The number of allowed bad logon attempts must be configured to three
 ↪  or less.
## /description
The account lockout feature, when enabled, prevents brute-force
 ↪  password attacks on the system. The higher this value is, the
 ↪  less effective the account lockout feature will be in protecting
 ↪  the local system. The number of bad logon attempts must be
 ↪  reasonably small to minimize the possibility of a successful
 ↪  password attack while allowing for honest errors made during
 ↪  normal user logon.
## /implementations/0/description
Configure the policy value for Computer Configuration >> Windows
 ↪  Settings >> Security Settings >> Account Policies >> Account
 ↪  Lockout Policy >> "Account lockout threshold" to "3" or fewer
 ↪  invalid logon attempts (excluding "0", which is unacceptable).
```

Listing 1: Example of a rule in a Windows-related security-configuration guide.

The publishers publish their recommendations on how to configure a software system in formats like PDF and in the Extensible Configuration Checklist Description Format (XCCDF), which is part of the Security Content Automation Protocol (SCAP). In some cases, these *implementations* are combined with machine-readable and automatable *checks*. These checks are created manually according to the specification written down in the security-configuration guides. Although XCCDF is designed as a machine-readable format, instructions for implementing the security settings are only contained in human-readable form in almost all cases. The notable exception is the OpenSCAP project's [16, 22] guides for Linux operating systems and applications, which for many rules contain shell scripts and parts of Ansible playbooks. Therefore, existing guides solve the lack-of-knowledge problem, but yield another problem: Automatic *implementations* (or remediation) are not specified in the SCAP standard. In contrast, the specification of automated *checking* is very detailed.

Publishers sometimes deal with this problem by providing additional artifacts, such as scripts or – in the case of Windows – configuration backup files. The problem here is threefold. Firstly, such artifacts do not exist for all guides. Secondly, the guides frequently get updated: If we take Windows 10 as an example, there will be at least one new guide every year published to deal with the updated settings, e.g., introduced by the version 1909 update; minor version updates deal with problems or changed requirements. As a result, DISA, for example, is now at version 18 for its Windows 10 guide. Therefore, creating/maintaining a mechanism (even if it can be based on some artifact provided by the publisher or) will be a recurring, manual task. Thirdly, with stand-alone artifacts for implementation, customization of guides, a feature which is central to SCAP, becomes cumbersome and error-prone, because this requires a manual effort to keep the customized guide in sync with the separately-maintained implementation mechanism. However, easy customization is essential: Experience shows that there is virtually no use case in which a publicly available security-configuration guide can be implemented without at least some changes.

The authoring process is depicted in Figure 1. The *publisher* creates the guide in the XCCDF format and the corresponding checks in the Open Vulnerability and Assessment Language (OVAL) format. This is a manual process, as the publishers incorporate their knowledge about the system and its architecture into the guide. In the next step, an *administrator* uses the automated checks to assess the state of their systems. The result is a list of the rules to which the system is not compliant; our evaluation in § 3 of over 2000 rules on systems using the default configuration shows that the rate of satisfied rules varies between 0% and 27%, with an average of 17.7%. Thus, for most of the rules, the (typically: default) configuration of the system to be hardened has to be adjusted.

If the publisher has not provided a mechanism for automated implementation, for every rule of this list, the administrator must read the implementation/remediation section of the rule in the XCCDF or PDF form of the guide and implement the steps described there. If a mechanism is provided, in most cases only a complete implementation of all configuration settings is possible. This creates significant manual effort for customization, especially if the implementation breaks functionality, but it is unclear which setting(s) have caused the observed problems.

In sum, we address one main *problem*: There are existing guides to configure systems securely, but we cannot implement the required configuration settings (taking into account necessary customization and changes due to updates of the guides) without significant manual effort.

Our *solution* to this problem, realized for Windows operating systems and applications, consists of three major steps. First, we process the files which define the Windows security policy settings that exist on a Windows-based system. Windows security policy settings are rules that administrators configure on a computer or multiple devices for the purpose of protecting resources on a device or network. [13] We can configure a policy setting with a policy path and a value. The so-called Administrative Template (ADMX/L) files define the majority of policy settings. They contain information about valid policy paths, possible values for each policy setting, and the underlying implementation of a policy setting within the Windows registry. Thus, we extract this knowledge in the first step and store it in a machine-readable format to access it during the remediation. Second, we use natural language processing to extract the settings and the intended values from the guides. We use the information of the first step to verify that the extracted setting exists and that the extracted value is a valid input for this setting and can, therefore, reduce the risk of wrongly extracted values to a minimum. Third, we translate the settings and values to their real implementation using the information from the first step. Our tools can use this information to implement as well as check the configuration settings automatically.

Our contributions are:

- an approach to how existing Windows-related security-configuration guides can be automatically implemented;
- a proof-of-concept implementation of our approach;
- a step-by-step documentation of our approach using the DISA Windows Server 2016 guide [26] and an updated version using the DISA Windows Server 2019 [28];
- an evaluation of our approach using existing guides from DISA and CIS with over 2000 rules [27].

In §2, we explain the general idea of our automatic implementation, and in the subchapters, we present the technical details of our proof-of-concept implementation. In §3, we use the DISA Windows Server 2016 guide and 12 CIS guides to demonstrate the feasibility of our approach. In §4, we discuss challenges and first experiences in generalizing our approach to non-Windows systems as well as additional future work. §5 treats related work and §6 concludes.

2 WINDOWS-RELATED SECURITY CONFIGURATION

Generic Approach. The generic approach is depicted in Figure 2. It shows the different stages of the envisioned process for automatically implementing Windows-related security-configuration guides. More specifically, the separate steps are defined as follows.

Extraction: Use natural language processing (NLP) for each rule to automatically extract the information needed to implement this rule.

Verification: Check with an automated mechanism that checks whether the derived information is valid:
- Does the extracted policy path indeed exist?

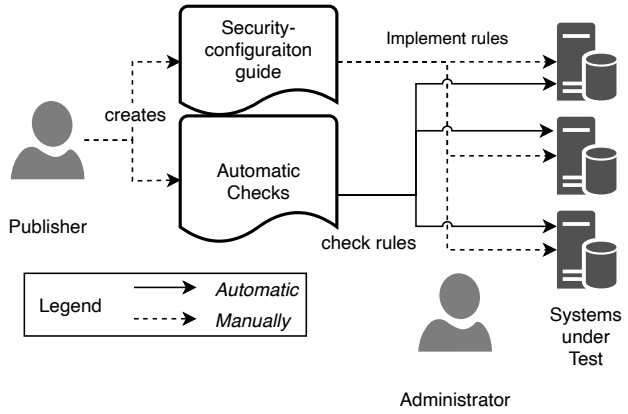

Figure 1: Current State of Implementation of Windows-related Security-Configuration Guides

  - Has the extracted value the required type for that setting?
  - Does the extracted value meet the requirements of that setting? Is it in the list of possible values or in the range of allowed values?

  If the path or the value is incorrect, the mechanism provides useful feedback about possible paths or values.

Transformation to low-level: Transform Windows policy settings into a representation of one of the underlying *low-level* implementation mechanisms. This step is necessary because almost none of the most popular configuration-management frameworks can directly process the Windows policy settings, but require the specification of an underlying implementation mechanism:
  - Registry settings
  - Secedit policy file entries
  - Audit file entries.

Transformation to code: Transform these low-level implementation mechanisms into code for carrying out the implementation of each setting.

Implementation: Execute code on the system we want to harden to implement the rules.

We emphasize that especially steps one and two are novel because – to our best knowledge – there is no approach published that uses NLP to extract policy settings from SCAP guides, nor is there an approach that verifies extracted values using definition files. For the evaluation of our approach, we assumed that an evaluation of the complete systems provides more evidence for the usefulness and feasibility of the presented approach than an evaluation of the first two steps alone. Consequently, we had to design and implement the remaining steps for our PoC implementation. In the end, we achieved the first published system that reads Windows-related security-configuration guides in the SCAP format and implements them automatically.

The approach in detail. We discuss the details of our approach and demonstrate its feasibility using a proof-of-concept (PoC) implementation.

```
system: org.scapolite.implementation.win_gpo
ui_path: <String containing a valid Windows policy path, using
          backslashes as separators>
value: <A YAML representation of a valid value for the
        specified path>
verification_status: (Checked. | Unchecked.)
```

Listing 2: Syntax of the Windows policy automation

The steps of our actual implementation, which we use as a proof of concept, are depicted in Figure 3. We describe them shortly here and more in detail in the rest of this section.

The input of our PoC consists of guides in the SCAP format. In the first step, we extract the necessary data for every rule to automate the implementation of this rule using natural language processing. The result is a set of rules enriched with the configuration settings in a machine-readable format. These configuration settings are then passed to the verification process: it has to be verified that the extracted data (a Windows policy path and required policy values) is valid. Our implementation uses the information of manually created verification rules for what essentially are legacy configuration settings combined with information extracted from the Windows administrative template files to verify the extracted values. To make the verification process as fast as possible, we process the latter files a priori and store the information we need in a database format.

If the verification is successful, the low-level automation needed to implement the rule is generated and also stored within the rule. Depending on the chosen implementation mechanism, these are used to create (1) either a group policy backup, which then can be imported on a Domain Controller to secure all systems in an Active Directory or (2) a JSON file used by a PowerShell script for implementing the settings. Additionally, our tooling can check the rules using the JSON files, but as SCAP already covers this aspect, we will not look deeper into this facet of our PoC.

In our PoC implementation, only the second and third steps require a minimum of manual interaction; the other steps are entirely automated. The dotted line between the *Verification* and the *Configuration Settings* in Figure 3 indicates that the person automating the security-configuration guide may have to execute the verification more than once and adjust the values until every rule is marked as *checked* by the verification mechanism.

In the following, we describe each of these steps. Tooling has been carried out in Python, except for a PowerShell framework for implementing and checking Windows security configurations using the output of Step 4. As a real-life example, we use the DISA Windows Server 2016 Security Technical Implementation Guide [7]. We created a GitHub repository [26], where we conducted all the steps, and created a commit and a tag after every step and reference them by their tags. [1]

[1]For representing the guide wihin Github, we use the YAML/Markdown-based *Scapolite* format developed within Siemens, which is better suited than SCAP for authoring and maintenance. The approach, though, is independent of the format.

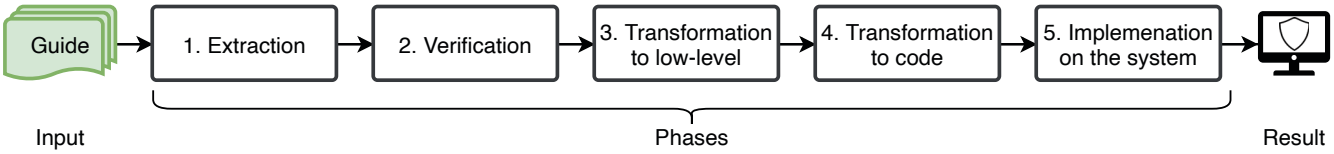

Figure 2: Overview of the abstract hardening approach.

2.1 Natural-language-processing-based extraction of Windows Policy Automations

The first step of our PoC implementation is the extraction of the needed values using NLP. Before we can extract the information needed to implement a Windows-related rule automatically, we had to define the structure of the machine-readable constructs,

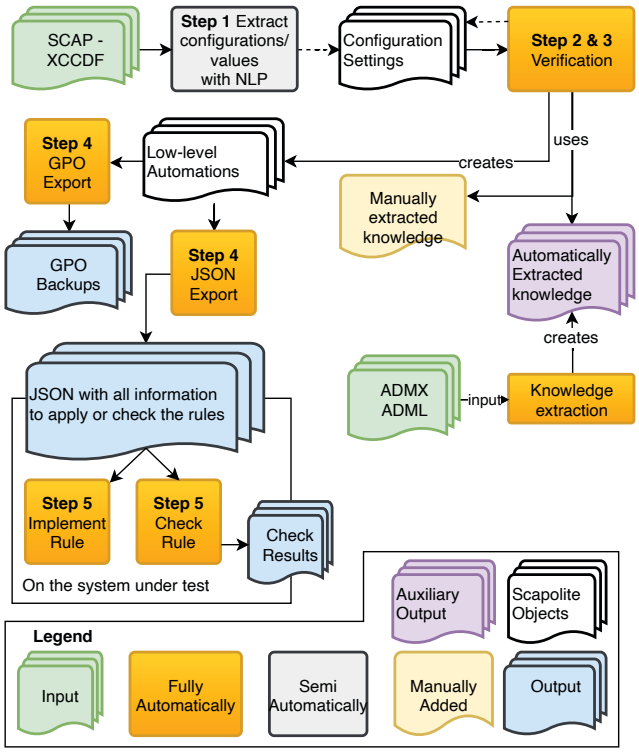

Figure 3: Overview of the steps of our actual implementation.

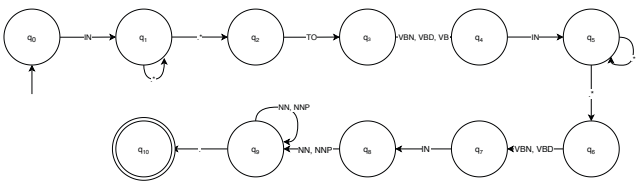

Figure 4: Example of an extraction rule as a nondeterministic finite automaton.

```
1  id: SV-88407
2  rule: <see below>
3  implementations:
4    - description: <see below>
5      automations:
6        - system: org.scapolite.implementation.win_gpo
7          ui_path: 'Computer Configuration\Policies\Windows
   ↪  Settings\Security Settings\Local Policies\User Rights
   ↪  Assignment\Back up files and directories'
8          value:
9          - Administrators
10 ---
11
12 ## /rule
13 The Backup files and directories user right must only be assigned to
   ↪  the Administrators group.
14 ## /implementations/0/description
15 Configure the policy value for Computer Configuration >> Windows
   ↪  Settings >> Security Settings >> Local Policies >> User Rights
   ↪  Assignment >> "Back up files and directories" to include only
   ↪  the following accounts or groups:
16 - Administrators
```

Listing 3: Example rule of the DISA Windows Server 2016 in YAML/Markdown form, incl. a Windows policy automation starting in line 6 (blue).

how they are integrated into the rule structure, and what has to be extracted to implement a rule.

For specifying Windows policy settings, the structure must provide information about the *policy path* and the required *value*. The type of the value (string, list, integer, et cetera) depends on the path; hence the specification of the automation syntax must refer to the set of valid Windows policy settings as shown in Listing 2. Listing 3 shows the usage of a policy automation in the rule *SV-88407* of the Windows Server 2016 guide. In an ideal scenario, the rule already contains the machine-readable automation objects, but this is not the case for guides published in the SCAP. Thus, we needed to extract the information about required policy settings from the human-readable description in the guide. To this end, we used the Natural Language Toolkit (NLTK) [1]. Due to the highly schematic structure of the guides under consideration, only eleven extraction rules had to be defined to process most of the rules. One of the rules is presented here in Listing 4 and Figure 4. The listing shows the definition of such an extraction rule as part of a grammar in NLTK. `IN`, `TO`, etc. refer to the corresponding part-of-speech (POS) tags. As we have ten rules, our grammar to extract the values consists of ten rule definitions. To make the idea more precise, Figure 4 is

```
SENTENCE_WITH_ENABLED_WITH_X_SELECTED_FOR_Y:
{<IN> <.*>+ <TO> <VBN|VBD|VB> <IN> <.*>+ <VBN|VBD> <IN> <NN|NNP>+
↪  <.>}
```

Listing 4: Example of an extraction rule with POS tags.

```
id: controlpaneldisplay__cpl_personalization_nolockscreencamera
registry:
  name: NoLockScreenCamera
  path: Software\Policies\Microsoft\Windows\Personalization
  hive: HKEY_LOCAL_MACHINE
  type: REG_DWORD
  enabled_value: 1
  disabled_value: 0
```

Listing 5: Example of a relationship between the id and the definition of the registry to set.

presenting the same rule as a nondeterministic finite automaton; q_0 marks the start state and q_{11} the end state.

We use NLTK to label the text of the description of a rule with POS tags. Afterward, the tagged sentences are passed to the grammar. If a sentence or a part of a sentence matches an extraction rule, then we know that here we can extract information for the automatic implementation. We use this sentence from rule *SV-92831* as an example: *"Configure the policy value for Computer Configuration » Administrative Templates » MS Security Guide » Configure SMBv1 client driver to Enabled with Disable driver (recommended) selected for Configure MrxSmb10 driver."* Now, we use NLTK to get the POS tags: *('Configure', 'VB'), ('the', 'DT'), ..., ('for', 'IN'), ('Computer', 'NNP'), ..., ('driver', 'NN'), ('to', 'TO'), ('Enabled', 'VB'), ('with', 'IN'), ('Disable', 'JJ'), ..., ('(', '('), (')', ')'), ('selected', 'VBN'), ('for', 'IN'), ('Configure', 'NNP'), ..., ('driver', 'NN'), ('.', '.')* The segment starting at *for* matches the pattern defined in the extraction rule, and we would reach the end state of Figure 4. Using our definition of the extraction rule, we know that we have the policy path in the part within the POS tags *IN* and *TO*, the first value between *TO* and *IN*, the second value between *IN* and *VBN, IN*, and the name of the option for which the second value has to be set between *VBN, IN*, and *'.'*.

As already mentioned, we need only eleven extraction rules to extract information for most of the DISA Windows Server 2016 guide; for a comparable CIS guide, we defined ten rules. Please note that the extraction using NLP is as simple as this only because DISA and CIS write their guides in a highly schematic way.

If the automatic extraction process could not obtain any or only ambiguous information for a setting to set, the respective rules are marked in this step of the process. For these rules, automation objects have to be created manually using the hints from the automatic extraction. For the analysis of the degree of automation, we refer to §3.1. Listing 3 is the result of a successful extraction carried out by our tool.[2]

2.2 Verification of Windows policy automations

As already mentioned in §2.1, the set of available policy settings determine the syntax (and semantics) of the Windows policy automations. The set of available policy settings varies between different versions of operating systems and policy-managed applications. Thus, we can determine the validity of a policy automation for a specific version of OS or an application. As mentioned before, the ADMX/L files define the majority of Windows policy settings. The Windows OSs use these files to display the GUI for configuring policy settings via point-and-click and keep the policy content and the actual implementation of the settings in the registry in sync. Microsoft regularly issues updates of the ADMX/L files.

To make this more visual, we provide another example: From the *ControlPanelDisplay.admx* and the *ControlPanelDisplay.adml* files located under policies on Windows Server 2016 instances, our exporter can get the information that the setting with the policy path *Computer Configuration \ ... Control Panel \ Personalization \ Prevent enabling lock screen camera* has the id *CPL_ Personalization_ NoLockScreenCamera*. We store this relationship and the information to which registry this id belongs in our export; this is presented in Listing 5. Here, we can get the information on which hive, path, and registry name are affected. Furthermore, we know that only *Enabled* and *Disabled* are valid options for this setting and that we can translate them to 1 and 0, respectively.

There are, however, also Windows policy settings that are not defined via ADMX/L files. These other settings are represented through entries in either a special configuration file (`GptTmpl.INF`) or a CSV file (`audit.CSV`) when creating a file-based representation of policy settings on a Windows OS through the `lgpo.exe` [12] tool provided by Microsoft. Unfortunately, there exists – to our best knowledge – no machine-readable representation that specifies these policy settings. Luckily, we could extract many of these specifications for configuration definitions from the SaltStack [23] implementation of the `win_lgpo` module for managing Windows configuration settings. (From the 196 settings configurable via the INF file, we could obtain 139 from SaltStack's implementation; the remaining specifications, which we encountered in the course of our work on several Windows OS versions, were added manually.) Furthermore, we could extract the specifications for all settings handled via `audit.CSV` via parsing a given `audit.CSV` file. Thus, the manual effort required for dealing with these non-ADMX/L settings was negligible when compared to the over 4000 configuration specifications we could extract automatically.

With the information of the knowledge extraction, the verification process can now determine for each configuration setting if the policy path is valid and, if so, whether the provided value is admissible for that particular policy path.

We have implemented our tooling such that the Windows policy automations in a given guide are parsed and verified. If the policy path exists and the given value is acceptable, the automation is marked as checked. If not, the automation is enriched with as much information as possible:

- If the policy path does not exist, information about similar policy paths is supplied, using the Levenshtein distance [11] on character and word basis over the set of valid policy paths. This set is a byproduct of our import step. To have the set of valid policy paths accessible is one reason to create those files a priori. Listing 6 a) provides an example of the result of the verification step.
- If the value is not admissible for the given policy path, information about admissible values is added to the automation – see Listing 6b) and c).

We proceed as follows to verify and correct the policy automations:

(1) The verification mechanism is run a first time.[3]
(2) The user reviews the reported errors and corrects them.

[2] Tag: step-3-extract-configurations-values-with-nlp

[3] Tag: step-4-verification-1

```
ui_path: ... \ Control Panel \ Personalization \ Prevent
    enabling lock screen
value: Enabled
error_class: NOT_FOUND policy name "preventenablinglockscreen"
error_hint: " The given path was not found, but there were 3 similar
 ↪  policies. If the UI path you were looking for is in the array,
 ↪  please replace the original UI path  with the new UI path."
candidates:
- Control Panel\Personalization\Prevent enabling lock screen camera
- ... \ Prevent enabling lock screen slide show
- ... \ Prevent changing the color scheme
---
ui_path: '... \ Network security: LAN Manager authentication level'
value: Send NTLMv2 response
error_class: CONFIGURE
error_hint: "To apply this rule, please choose a setting value for
 ↪  each sub-setting in candidates. Next, replace the content of the
 ↪  'value' attribute with the content of candidates."
candidates:
- Send LM & TLM responses - use NTLMv2 session security if negotiated
- Send NTLMv2 response only. Refuse LM & NTLM
- Send NTLM response only
...
---
ui_path: ... \ Configure Windows Defender SmartScreen
value: Enabled
candidates:
    main_setting:
      - Disabled
      - Enabled
    Pick one of the following settings:
      - Warn
      - Disabled
      - Warn and prevent bypass
```

Listing 6: Failed verifications: a) Policy path does not exist; information about 3 possible options. b) Specified value does not exist; admissible values provided. c) Policy setting underspecified; request for additional value.

(3) Verification is re-run either on a rule-by-rule basis or for the complete guide.[4]
(4) Once all errors have been corrected, an export pairing the human-readable description and the policy automation for each rule is created, allowing the user to verify very quickly that the automation indeed faithfully reflects the human-readable specification.[5]

This verification seems simple, but studies have shown that 42% of the configuration errors that caused high-impact incidents are obvious errors (e.g., typos) [31] and that a significant number of configuration errors are due to compatibility issues[40]. Our verification is able to catch such problems at the earliest possible stage.

2.3 Generation of low-level implementation mechanisms

Windows policy settings are implemented through registry settings, *INF* policy file entries, and audit file entries. To represent these mechanisms within a guide, we introduce automation extensions for these three mechanisms. Using the information gathered as described in §2.2, we implemented a transformation from the policy automation into the corresponding *low-level* automation extension.[6]

[4]Tag: step-4c-fix
[5]Tag: step-5-create-xlsx-report-for-the-current-guide
[6]Tag: step-6-enrich-scapolite-with-low-level-automations, a table with all the low-level automations can be found under xlsx/report_with_low_level_automations.xlsx.

```
ui_path: ...\Apply UAC restrictions to local accounts on network
 ↪  logons
value: Enabled
verification_status: Checked.
- system: org.scapolite.implementation.windows_registry
  config: Computer
  registry_key:
   ↪  SOFTWARE\Microsoft\Windows\CurrentVersion\Policies\System
  value_name: LocalAccountTokenFilterPolicy
  action: DWORD:0
```

Listing 7: Example of a Windows policy automation and the resulting Windows registry automation.

Listing 7 provides an example: according to the Windows policy automation (line 1 to 4), the value *Enabled* has to be set for the policy setting with path ... \ *Apply UAC restrictions to local accounts on network logons*. Using information extracted from the ADMX/L files, we can generate the Windows registry automation: the registry key under the path *SOFTWARE \ Microsoft \ Windows\CurrentVersion\Policies\System* with the value name *LocalAccountTokenFilterPolicy* has to be set to a *DWORD* with the value 0.

2.4 Transformation into code

The main idea between the separation of this step and the actual implementation was that we could execute all the previous steps on one machine, export the information, and do only the actual implementation on the system under test. Thus, a central instance can be used for storing and processing relevant guides; systems under test can fetch the required data for implementing (and testing) security configurations from that central server. To further facilitate this approach, we implemented an export from a guide containing *low-level* automation for Windows into a JSON document that contains all data relevant for implementing each rule with the associated automation(s).

In order to support implementations via policies (either via local group policies or via the Active Directory capabilities), we can also automatically generate policy backups based on the extracted information. We have implemented this step as part of a continuous-integration where changes to automation in a guide lead to an automated re-generation of both scripts and policy backups.[7]

2.5 Implementation of the rules on the system using PowerShell

When choosing a target language framework to use to implement the rules using the information of the tow-level automation described in §§2.2 and 2.3, we decided to use PowerShell for the following reasons:

- Common configuration management frameworks like Ansible, Puppet, Chef, and SaltStack cannot handle the Windows policy settings or use PowerShell to implement them. Thus, we decided to use PowerShell without a configuration framework as a wrapper to implement the rules.
- Microsoft's efforts to allow code/script-based configuration management of Windows rather than the GUI-based mechanism centering on the policy editor are based on PowerShell.

[7]Tag: step-8-export, policy backup folder for each profile under lgpo_backups.

Table 1: Extracted, verified, and automated rules.

Categories	#	%	% of OVAL
Rules	274	100	
Configurations Extracted with NLP	198	72.3	95.6
Rules without extracted values	76	27.7	36.7
First-Time Verified	173	63.1	83.6
Not verified the first time	25	9.1	12.1
Non-automatable but extracted	2	0.7	1.0
Automatable but not extracted	4	1.5	1.9
Verified after manual correction	27	9.9	13.0
Automated Rules	200	73.0	96.6

- PowerShell is installed by default on all Windows OSs that are still in mainstream support by Microsoft.
- To fully leverage the ability to generate mechanisms for rule-by-rule implementation rather than the *bulk* implementation offered, e.g., in the form of policy backups, we looked for robust roll-back functionality that allowed us to reset a configuration reliably to its previous value.

Thus, we have created a PowerShell library that, based on the JSON file, applies, checks, and reverts single as well as several or all rules. As mentioned before, our tooling uses the extracted information to check whether the system is compliant to a rule automatically. This functionality is already covered within SCAP, and there are many SCAP-compliant scanners. Therefore, the checking functionality is not in the focus of this paper.

Our PowerShell library uses Windows tools that assure that the configuration changes are reflected in the local policy: *secedit*, *auditpol*, and *LGPO.exe* [12]. In the end, we can implement a security-configuration guide by running one PowerShell command.

3 EVALUATION

To demonstrate the presented approach's potential, we use the real-life example of realizing automatic rule-by-rule implementations for the DISA Microsoft Windows Server 2016 guide Benchmark [7] for an evaluation.[8] The benchmark consists of 207 rules with automatic checks and 67 rules without automatic checks.

The results of all steps shown below are available for review [26]. Every step is denoted as a commit and marked with a tag. Thus, a diff view between a commit and its predecessor reveals the constructs added, removed, or changed in this step. In this article, we will concentrate on this repository. Additionally, we created a new repository with the DISA Windows Server 2019 guide[28] and executed the same steps to demonstrate that our approach works on recent SCAP documents as well. Thus, the fact that we used Windows Server 2016 should not be a threat to our evaluation's validity.

We seek to answer the following **Research Questions**:

RQ1 For how many rules can we automatically derive an implementation from the text in natural language? How high is their percentage?

RQ2 How many of the extracted rules are automatable, and how many automatable rules were not extracted?

RQ3 After correcting wrongly extracted automations: How many rules can we implement automatically for the complete guide?

RQ4 How much time does our approach require to extract the information, verify it, and implement the rule?

RQ5 How many rules are implemented correctly in accordance with the automated checks?

We will use the DISA Windows Server 2016 guide to answer **RQ1-4** and several CIS guides to answer **RQ5**. For **RQ5**, we use CIS guides because, for them, we have the automatic checks and can assess a given system using their CIS-CAT tool.

3.1 Degree of automation

To answer **RQ1**, **RQ2**, and **RQ3**, we examine the steps regarding the extraction of Windows policy automation using NLP and the verification of the found policy paths and values. The results are depicted in Table 1. From the 274 rules in the Windows Server 2016 guide, we can extract for 198 rules a possible policy setting with possible values. Afterward, from the 198 possible configuration path/value pairs, 173 can be directly verified as valid configuration settings by the first verification step. These 198 rules mean that for 63% of the rules, we can extract both the policy path and the required value and verify that this value is valid for the particular policy path without any manual effort. Thus, we could answer **RQ1**. From the remaining 25 rules, for two rules, potential configuration settings and values have been extracted erroneously: with our automation mechanisms, we could not automate these two rules. We removed the erroneously created automations for theses two rules manually.[9] Conversely, for four rules that we could automate, neither the policy path nor the required value was extracted. In this case, we added the automation manually.[10] Thus, the ratio of rules not added to the set of rules to automate, although they are automatable, lies at 1.5%, whereas the ratio of rules which are not automatable and still extracted is 0.7% regarding all rules. For the remaining 23 rules that were extracted but could not be verified in the first round, we created the correct automation based on the extracted information enriched with the verification process's hints.[11]

If one sees the NLP based extraction process as a classifier with the classes *automatable* and *non-automatable*, the false-positive rate of this classifier is at 2.7% and the false-negative rate at 2.0%. We had to adjust 27 rules manually. Thus, for 90.1% of all rules, respectively, 87% of the automatable rules, no manual action was needed throughout the process. In summary, these numbers answer **RQ2** and give strong evidence for the importance of our verification step because otherwise, these rules might be applied wrongly or not at all.

After the execution of the extraction and the verification step and the manual adjustments, we now have 200 rules which can be

[8] We choose DISA's guide because their SCAP content is public. Only CIS members can access CIS's SCAP content, whereas their PDFs are publicly available.

[9] Tag: step-4a-fix-rules-which-have-been-imported-but-are-not-automatable

[10] Tag: step-4b-fix-rules-which-have-not-been-imported-but-are-automatable

[11] Tag: step-4c-fix

Table 2: Time needed to execute the single steps with all 200 automatable rules of the DISA Windows Server 2016 guide.

Step	Time (s)
Knowledge Extraction from ADMX/L	81.59
Import into Scapolite	8.02
NLP extraction of policy automations	16.93
Verification of policy automations	23.48
Export automations in JSON	13.90
Export automations in XLSX	14.03
Export policy backups from JSON	1.65
Check all rules for compliance	13.96
Implement automatable rules one-by-one	73.35
Σ	245.91

automated and have values that are verified to be valid for the given configuration decisions. Therefore, the grade of automation we can achieve on the set of the 274 rules is at 73.0%, respectively, at 96.6% if we are only considering the 207 automatable rules (classified as automatable by DISA). This number answers **RQ3**. Thus, our approach reduces the number of rules which have to be checked or set manually on the system under test significantly.

3.2 Time

Table 2 shows time values for each of the automated steps.[12] The short execution time per rule enables an application in CI approaches, which answers **RQ4** partially.

If we want to calculate the overall time, we also have to include the time it takes to correct the wrongly extracted automations. According to Table 1, 25 rules were not verified the first time. Because of the feedback included in the rule, we assume that it takes 10s to correct such a rule. For the remaining four plus two rules, we assume that it takes at most 2min per rule to correct it. These assumptions are also backed by the feedback of the users of our tools at Siemens. Therefore, we end up with a total time of $245.91s + 25 * 10s + 6 * 120s = 1215.91s \approx 20min$ for all rules or $6s$ per rule. Thus, **RQ4** could be answered, too.

3.3 Correct Application

In the last step of our evaluation, we want to answer whether our approach is applying the security-configuration guides correctly. Incorrectly implemented rules can result from faults in the ADMX/L importer, the verification process, or the PowerShell library. Here, our idea was that after applying a security-configuration guide to a system, the system should be configured as specified in the guide. For this experiment, we use the standardized OVAL checks as ground truth. Thus, we used guides which are Windows-related and for which we have automated checks. Therefore, we used in this step 12 different security-configuration guides from the CIS, which are listed in Table 3, totaling over 2000 rules: Four Windows-based OS's, six components of the Office package, and two browsers.

We conducted the evaluation as follows: First, every security-configuration guide was automated through the same process, as explained in §2. Next, we set up a clean environment for every system.[13] Additionally, we installed a SCAP-compliant scanner on the machines, i.e., the CIS-CAT tool [4]. Next, we executed the checks in the *clean* environment to compare the clean state with the hardened state to show that the implementation of guides makes the system more secure. Afterwards, the guides are implemented using the automation generated as described in §2. Now the checks are rerun to test whether the implementation was correct. The results are depicted in Table 3. We also published the check reports before and after the implementation on GitHub [27]. Within this repository, one can find for every checked guide a *before.html* and *after.html* containing the result of the automatic check created using the CIS-CAT tool.

Note that we only consider the rules which have OVAL checks for the calculation of the percentages. We see that for the OSs between 16.9 and 26.3% of the rules are already set up in a compliant way, whereas nearly no rule is pre-configured securely for the other components. Nevertheless, even 26.3% of already fulfilled rules of the OSs imply that the majority of the settings are configured in an insecure way on a clean system. After applying the rules, the percentage of compliant rules is between 95 and 100% for all guides.

That we do not reach 100% compliance relative to the results of the CIS-CAT checker tool is due to errors in the guides, some of them in the automated check, others in the descriptive text. For example, some checks are overspecified, i.e., they expect more changes than actually occur when implementing the corresponding configuration setting: the rule *18.5.9.1* of the Windows 10 benchmark changes only a single registry entry, but the corresponding check refers to three different entries. Also, some rules have automatic checks which test for wrong values. For example, the check for the rule *1.8.7.4* of the Word guide expects a different value (namely 0) than the value, which is set if the rule is implemented manually following the security-configuration guide. Thus, we have in this rule precisely the difference of implementation and check we want to overcome with our approach. Finally, there were some errors regarding the description of the implementation provided in the guides. For example, rule *1.8.7.2.7* of the Word guide specifies that the setting should be enabled, although title and description suggest disabling the setting. Another error in a guide actually is due to a misspelling of the ADMX/L template file provided by Microsoft. For example, rule *1.13.2.1.5* of the Outlook guide specifies the value to be implement as *When online always retrieve the CRL*, but our tool could not validate this value for this setting because of a misspelling in a template file. There, the value is written as *When online always retr**ei**ve the CRL*.

All in all, we achieved compliance for 1965 rules (i.e., 97.6%) after implementing the guides. For the OSs, we have the highest absolute gain of compliant rules (between 237 and 404 rules), but in relative numbers, we are only gaining between 71% and 80%,

[12] All the steps are conducted by running different commands from the command-line. We ran every command 50 times and averaged the elapsed time to evaluate the speed of the single steps. Configuration: 3.1 GHz Intel Core i7 with 16 GB RAM, Python 3.7.4. The only steps implemented in PowerShell are the application viz. the check for compliance step as these were designed to be executed on the system under test, in our case, a Windows-based system, without installing any additional software. PowerShell Version 5.1.14393 was used.

[13] For the OS's, we have set up every system in a new VM by installing the OS directly from the latest ISO down-loadable from Microsoft. As a VM provider, we used VirtualBox. For the other components, e.g., Chrome or PowerPoint, they were directly installed on a clean Windows 10 instance.

Table 3: # rules per guide compliant to the given guide before and after implementing guide automatically. Highest value of a column: dark gray, lowest: light gray.

Guide	# of Rules	OVAL	Before	%	After	%	Δ	Δ %
Google Chrome for Windows	20	20	0	0	19	95.0	19	95.0
Internet Explorer 11	156	136	1	0.7	132	97.1	131	96.3
Microsoft Office	53	53	2	3.8	52	98.1	50	94.3
Microsoft Access	9	9	0	0	9	100	9	100
Microsoft Excel	34	34	0	0	34	100	34	100
Microsoft Outlook	75	73	3	4.1	72	98.6	69	94.5
Microsoft PowerPoint	18	18	1	5.6	18	100	17	94.4
Microsoft Word	24	24	0	0	23	95.8	23	95.8
Windows 7	390	386	87	22.5	377	97.7	290	75.1
Windows 8.1	429	425	90	21.2	415	97.6	325	76.5
Windows 10	505	502	85	16.9	489	97.4	404	80.5
Windows Server 2016	371	334	88	26.3	325	97.3	237	71.0
Σ	2084	2014	357	17.7	1965	97.6	1608	79,8

whereas for the rest, we have a gain of over 90%. Please note that our approach can also implement the settings which were already compliant on a clean instance, but we have chosen this scenario because it seemed more relevant and natural. The alternative would have been to create an instance in which every setting is configured to a non-compliant value.

Discussion. In **RQ1**, we asked for the percentage of rules for which we can automatically extract the implementation. If our approach extracted the implementation only for a small fraction of rules, it would be useless in real-world applications. Since we extracted for 63% of all rules and 96% of automatable rules an implementation, we can rule out this concern.

In **RQ2**, we looked for the percentage of false negative and false positives of our extraction process. If these numbers were too high, the administrators would spend much time identifying them so that our approach would become pointless. With 1% and 2% of the automatable rules wrongly classified, this is not the case.

In **RQ3**, we searched for the percentage of rules that we can automate after correcting the extraction process's errors. If this number were too low, administrators would spend the same amount of time for implementing the remaining rules, and the gains of our approach would be small. Our results show that we can automate 97% of the automatable rules with our approach and dramatically reduce manual implementation.

In **RQ4**, we asked for the time taken to execute our approach. If the steps were too time-consuming, it would be more efficient to do it manually, and our tooling would be unnecessary. With 245.91s for the tools themselves and 1215.91s for the complete process, our approach is more efficient than the manual approach.

In **RQ5**, we searched for the percentage of rules which are correctly implemented according to the automatic checks. If our approach implemented the rules wrongly, it would be useless. With over 97% of correctly implemented rules, our approach implements almost all rules correctly.

In summary, our evaluation showed that our approach is feasible and effective.

4 GENERALIZATION AND FURTHER WORK

The main limitation of our extraction step is the fact that this extraction is only possible because of the highly schematic structure of the descriptions written by CIS and DISA. If they modify their template for these descriptions, we will have to adjust this step entirely. Thus, we hope that future guides will have the needed information in a machine-readable form. A limitation of our implementation of Windows-related guides is the dependency on the *LGPO.exe*. If Microsoft decided to remove this tool for changing Windows system settings, we would have to replace core parts of the presented approach.

We admit that our approach is only an intermediate solution. Instead of converting guides to executables by users or third parties, it would be more practical for publishers to attach machine-executable codes or links to them to the rules as they are doing it for automatic checking. Nevertheless, as long as the publishers do not distribute the guides so that we can quickly and automatically implement them, we need tools like those we presented in this paper.

```
## /implementations/0/description
Follow the below steps to disable `Location Services`:
1. Tap `Settings` Gear Icon.
2. Tap `Security & Location`.
3. Scroll to the `Privacy` section.
4. Tap `Location`.
5. Toggle to the `OFF` position.
```

Listing 8: Example of an implementation as part of a rule in an Android security-configuration guide.

```
ui_name: Location
namespace: secure
name: location_providers_allowed
value:
 ON: +network,+gps
 OFF: -network,-gps
```

Listing 9: Example of a definition for an Android-related setting.

```
## /implementations/0/description
Set the following parameters in `/etc/sysctl.conf` or a
`/etc/sysctl.d/*` file:
```
net.ipv6.conf.all.accept_ra = 0
net.ipv6.conf.default.accept_ra = 0
```
Run the following commands to set the active kernel parameters:
```
sysctl -w net.ipv6.conf.all.accept_ra=0
sysctl -w net.ipv6.conf.default.accept_ra=0
sysctl -w net.ipv6.route.flush=1
```
```

Listing 10: Example of an implementation in an Ubuntu Linux security-configuration guide.

Our approach is tailored to Windows and its policies. Thus, the approach cannot be ported to other platforms without significant adjustments. Nevertheless, we are developing similar approaches for Linux OSs and Android in particular and try to achieve similar results there as well.

In Listing 8, one can see the implementation of an Android-related rule. It describes highly schematically the actions to implement. Thus, the difficulty of the *extraction* process as described in Figure 1 is comparable to that for the Windows-related guides.

The *verification* step is more difficult, because we do not have a similar definition of potential settings and the set of values they can have. In Windows, we can extract this information from the ADMX/L files, but in Android, there are – to our best knowledge – no comparable files available. To port our approach to Android, we created such definition files for several settings. For the setting *Location*, one would find an entry in this definition file as presented in Listing 9. With this information, we can verify that *OFF* is a valid value for this setting. Furthermore, we can use the information that we can translate *OFF* to *-network,-gps* for the *transformation to a low-level automation*. Finally, we can implement the rule on a given Android device via the Android Debug Bridge and the translated value.

Our work on Android just started, and there are many open questions: How could be the syntax of an Android definition file? How can we automatically create such a definition file? Which settings can we automatically set, e.g., via the Debug Bridge, and which settings cannot be set or only if we have rooted the device? How can we handle different Android versions and the fact that we can automatically configure a setting in one version, e.g., via the Debug Bridge, and in another version, it is no longer possible?

For the automated implementation of general Linux guides, please have a look at Listing 10; here, we have the implementation of a rule of a Ubuntu guide. We can see that there is still a schema of how the implementation is described. Nevertheless, it is more complicated. In this example, there are two different steps, one concerning the modification of a file, the other the execution of shell commands. Hence, in addition to extracting the code-snippets, we have to derive the semantics of *set file content to* and *run* as well. If we wanted to *verify* that the code snippets are valid, we would have to know the syntax of the specific configuration file and the semantics, e.g., if *net.ipv6.conf.all.accept_ra = 0* is a correct line in this file. Furthermore, we would have to know the legal parameters of the program called in the second snippet.

In our future research, we will try to extract this information, e.g., from the source code, the documentation, or sample configuration file to create definition files for the most common commands and configuration files. In summary, we think that our approach can be ported from Windows to Linux-based systems, but whether a comparably high percentage of rules from which automations can be extracted can be reached remains to be seen.

In the future, work is necessary to provide the foundations that make security automation easier. The main factor that made our approach possible was that Microsoft provides machine-readable information about configuration options for their systems in the form of ADMX/L files. It follows that vendors should support security automation by providing machine-readable information about security-configuration options and their implementation.

5 RELATED WORK

Many studies have been conducted in the field of misconfiguration, e.g. [5, 6, 8, 31, 38]. Especially the study of Dietrich et al. [6] is relevant for our research. Their study provides strong evidence that security misconfigurations are more common than usually assumed. This emphasizes how important and yet underestimated this field of research currently is. Furthermore, they have identified the lack of knowledge and experience as core factors for security misconfiguration and argue that we need more automation in the whole process to make systems more secure. By using security-configuration guides, we want to tackle the first problem, with our automated implementation the second.

Additionally, many researchers explored how to detect and how to avoid misconfigurations [10, 21, 24, 29]. Rahman et al. [21] analyzed thousands of Infrastructure-as-a-Code (IaaC) scripts to identify insecure configurations and security smells. They used these smells to create a linter for creating more secure IaaC scripts. Although their linter is comparable to the hints we give to the administrators, we are targeting different problems. Where they are extracting knowledge from the IaaC scripts on how to configure systems securely, we already have this information and have to apply it. Furthermore, as discussed before, we think that IaaC scripts are not sufficient to specify security-configuration guides. Similar work was done by Santolucito et al. [24]. Their framework *ConfigV* aims at similar problems as our verification step. In contrast to them, we cannot learn secure configurations. Instead, these are defined in the guides, and the constraints do not have to be learned but can be extracted from the ADMX/L files. Similarly, SPEX, developed by Xu et al. [37] is not applicable in our case, as we do not have the source code of the programs we want to configure.

Raab et al. [17–19] created the *Elektra* framework to validate the access to configuration values to detect misconfigurations as soon as possible. We tried to achieve the same with our a-priori verification process. In their study [19], they investigated how free/libre and open-source software (FLOSS) can be configured and the problem of validating configurations for it. One finding is that presently, configuration validation is encoded in a way unusable for external validation or introspection tools. Although Windows is not a FLOSS, we encountered the same problem. This is why we had to implement our verification mechanism instead of simply using an existing tool. Furthermore, Elektra is tailored towards developers who create new

software, not for administrators of existing software and it cannot handle the Windows policy settings we have to change according to the guides. Thus, we could not apply Elektra.

A similar approach to Elektra was developed by Xu et al. [36] with the same problems so we could also not use it in our case. In their study [35], they have shown how the growing complexity of the configuration of systems is overwhelming users and systems administrators. They did not investigate Windows systems, yet many of their findings apply to our domain, too. For instance, users have tremendous difficulties because they do not know which parameters to set and that this induces up to 50% of the configuration errors. This supports the claim that we need security-configuration guides created by experts, to be used by system administrators.

Wang et al. [33] present an approach at automatic reverse engineering of an application's access-control configurations. Although the application domain is similar to our context, we could not apply their work for our need as we do not have the source code of the programs we want to configure securely.

There also is a lot of research in the field of extracting important parameters or configuration values from human-readable documents [9, 15, 20, 25, 30, 34, 39, 41]. Yang et al. [39] present an approach to automatically extract web API specifications from the documentation of a software similar to the extraction of our configuration values from the security-configuration guidelines. However, the fact that our documents do not contain as many links made this approach unfeasible in our case. Using NLP, Wong et al. [34] developed an approach at extracting information from program documentation to improve automated testing. They use grammar rules to identify relevant comments and extract constraints from them. In our case, the security-configuration guides describe concepts from a higher level then program documentation. Furthermore, we do not need to extract the constraints from the security-configuration guide. Thus, this approach was also not applicable in our context.

Closest to our work regarding our aims of providing rule-by-rule implementation is the OpenScap project [16, 22]. OpenScap maintains its security-configuration guides for various Linux systems in a git repository, where each rule is represented by one file; usually, the file holds references to other files containing artifacts for automated implementation and check. However, we cannot use OpenScap. First, OpenScap cannot implement the rules from the Windows-based guides. Second, if OpenScap could implement them, we would first have to add the scripts manually to the guides of CIS or DISA. We think that the guides in the context of OpenScap are one step ahead of Windows guides published by CIS or DISA because of the connection between implementation and checking. In the future, we hope that the publishers distribute their Windows guides similarly to OpenScap in a form that is as easily implementable as checkable. We consider our approach an intermediate solution to bring the automatic implementation of Windows-based guides to a comparable level as long as this is not the case.

Ongoing activities regarding further improvements of automating security as carried out by the IETF SACM (*Security Automation and Continuous Monitoring*) work group [2, 3, 14] as well as a first indication of the direction work towards SCAP version 2 as outlined in a transition document [32], have a clear focus on checking security-configuration settings and disregard their implementation—which is precisely the gap we want to close in this work.

To sum up the related work: some approaches use NLP to extract settings from the documentation or the source code of a program, but to our best knowledge, no approach extracted the settings from security-configuration guides. Furthermore, some approaches like Elektra help to improve the configuration of newly developed software, but we cannot use them to configure existing and closed-source Windows systems. We can automatically implement the guides of some Linux variants with the OpenScap approach if the publishers distribute them with the scripts necessary for OpenScap. However, we cannot use OpenScap to implement existing Windows-based guides automatically. Thus, we tackled these gaps in the literature and put the developed components together to demonstrate that our proposed approach and our PoC implementation are achieving promising results.

6 CONCLUSION

The complexity of contemporary systems renders their configuration increasingly difficult. This leads to vulnerabilities attackers can exploit to attack the systems. For a single organization, it is impossible to know all the configurations to make a specific system secure. Many organizations use public security-configuration guides to overcome the lack of knowledge; while many of these guides support automated compliance checking, they do not provide support for automated implementation.

In this paper, we demonstrated an approach that can automatically implement Windows security-configuration guides with minimal manual effort. Our contribution further encompasses a proof-of-concept implementation, a step-by-step documentation of the process, and the evaluation of our approach using existing guides.

Our evaluation has shown that we can automate 83% of the rules without any manual effort using our NLP extraction. Furthermore, our extensive benchmark with 12 different guides and over 2014 rules with automatic checks showed that the implementation of our approach can implement at least 97% of the rules correctly.

With our approach and the results of its evaluation, we believe we can furthermore contribute as follows: Firstly, we have demonstrated how organizations that rely on publicly available security-configuration guides can be aided in reducing effort as well as reducing errors in the implementation of these guides. Secondly, we have shown how machine-readable information supporting automated implementation for Windows systems can be represented and included in SCAP guides. We hope that our results encourage publishers of security guides to support better the automated implementation of their guides by enriching them with such information, for Windows as well for other target systems. The design of SCAP v2 has already started [32]: Our work offers timely and relevant input for the further development of SCAP towards a standard that meets the requirements of both publishers and consumers of machine-readable security-configuration guides.

Thirdly, our research underlines the need for machine-readable specifications of (security) configuration settings: standardization and support of a format for this purpose by vendors would significantly aide in all tasks concerned with configuring systems securely.

We plan to release substantial parts of our Python code-base.

REFERENCES

[1] Steven Bird, Ewan Klein, and Edward Loper. 2009. *Natural language processing with Python: analyzing text with the natural language toolkit. "* O'Reilly Media, Inc.".

[2] Henk Birkholz, Jarrett Lu, John Strassner, Nancy Cam-Winget, and Adam W. Montville. 2018. *Security Automation and Continuous Monitoring (SACM) Terminology.* Internet-Draft draft-ietf-sacm-terminology-16. Internet Engineering Task Force. https://datatracker.ietf.org/doc/html/draft-ietf-sacm-terminology-16 Work in Progress.

[3] Nancy Cam-Winget and Lisa Lorenzin. 2017. Security Automation and Continuous Monitoring (SACM) Requirements. RFC 8248. https://doi.org/10.17487/RFC8248

[4] CIS. 2019. *CIS-CAT Pro.* https://www.cisecurity.org/cybersecurity-tools/cis-cat-pro/

[5] Andrea Continella, Mario Polino, Marcello Pogliani, and Stefano Zanero. 2018. There's a Hole in That Bucket!: A Large-scale Analysis of Misconfigured S3 Buckets. In *Proceedings of the 34th Annual Computer Security Applications Conference* (San Juan, PR, USA) *(ACSAC '18)*. ACM, New York, NY, USA, 702–711. https://doi.org/10.1145/3274694.3274736

[6] Constanze Dietrich, Katharina Krombholz, Kevin Borgolte, and Tobias Fiebig. 2018. Investigating System Operators' Perspective on Security Misconfigurations. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. ACM, New York, NY, USA, 1272–1289. https://doi.org/10.1145/3243734.3243794

[7] DISA. 2019. DISA Microsoft Windows Server 2016 STIG Benchmark. Available from https://dl.dod.cyber.mil/wp-content/uploads/stigs/zip/U_MS_Windows_Server_2016_V1R13_STIG_SCAP_1-2_Benchmark.zip. Accessed: 2019-01-22, we used the version 7, current version is 13.

[8] A. K. Jha, S. Lee, and W. J. Lee. 2017. Developer Mistakes in Writing Android Manifests: An Empirical Study of Configuration Errors. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. 25–36. https://doi.org/10.1109/MSR.2017.41

[9] Dongpu Jin, Myra B. Cohen, Xiao Qu, and Brian Robinson. 2014. PrefFinder: Getting the Right Preference in Configurable Software Systems. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering* (Vasteras, Sweden) *(ASE '14)*. ACM, New York, NY, USA, 151–162. https://doi.org/10.1145/2642937.2643009

[10] L. Keller, P. Upadhyaya, and G. Candea. 2008. ConfErr: A tool for assessing resilience to human configuration errors. In *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*. 157–166. https://doi.org/10.1109/DSN.2008.4630084

[11] Vladimir I Levenshtein. 1966. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, Vol. 10. 707–710.

[12] Microsoft Corporation. 2016. *Local Group Policy Object Utility.* https://www.microsoft.com/en-us/download/details.aspx?id=55319 Accessed: 2019-01-18.

[13] Microsoft Corporation. 2017. Security policy settings. https://docs.microsoft.com/en-us/windows/security/threat-protection/security-policy-settings/security-policy-settings

[14] Adam W. Montville and Bill Munyan. 2018. *Security Automation and Continuous Monitoring (SACM) Architecture.* Internet-Draft draft-ietf-sacm-arch-00. Internet Engineering Task Force. https://datatracker.ietf.org/doc/html/draft-ietf-sacm-arch-00 Work in Progress.

[15] Rahul Pandita, Xusheng Xiao, Hao Zhong, Tao Xie, Stephen Oney, and Amit Paradkar. 2012. Inferring Method Specifications from Natural Language API Descriptions. In *Proceedings of the 34th International Conference on Software Engineering* (Zurich, Switzerland) *(ICSE '12)*. IEEE Press, Piscataway, NJ, USA, 815–825. http://dl.acm.org/citation.cfm?id=2337223.2337319

[16] Martin Preisler and Marek Haicman. 2018. Security Automation for Containers and VMs with OpenSCAP. In *USENIX LISA*. Washington. Available from https://martin.preisler.me/...

[17] Markus Raab. 2015. Safe Management of Software Configuration. In *Proceedings of the CAiSE'2015 Doctoral Consortium at the 27th International Conference on Advanced Information Systems Engineering (CAiSE 2015), Stockholm, Sweden, June 11-12, 2015.* 74–82. http://ceur-ws.org/Vol-1415/CAISE2015DC09.pdf

[18] Markus Raab. 2016. Improving system integration using a modular configuration specification language. In *Companion Proceedings of the 15th International Conference on Modularity, Málaga, Spain, March 14 - 18, 2016.* 152–157. https://doi.org/10.1145/2892664.2892691

[19] Markus Raab and Gergö Barany. 2017. Challenges in Validating FLOSS Configuration. In *Open Source Systems: Towards Robust Practices - 13th IFIP WG 2.13 International Conference, OSS 2017, Buenos Aires, Argentina, May 22-23, 2017, Proceedings.* 101–114. https://doi.org/10.1007/978-3-319-57735-7_11

[20] A. Rabkin and R. Katz. 2011. Static extraction of program configuration options. In *2011 33rd International Conference on Software Engineering (ICSE)*. 131–140. https://doi.org/10.1145/1985793.1985812

[21] Akond Rahman, Chris Parnin, and Laurie Williams. 2019. The Seven Sins: Security Smells in Infrastructure As Code Scripts. In *Proceedings of the 41st International Conference on Software Engineering* (Montreal, Quebec, Canada) *(ICSE '19)*. IEEE Press, Piscataway, NJ, USA, 164–175. https://doi.org/10.1109/ICSE.2019.00033

[22] Red Hat, Inc. 2010. OpenSCAP. https://www.open-scap.org. Accessed: 2018-12-18.

[23] SaltStack, Inc. 2011. *SaltStack.* https://github.com/saltstack/salt Accessed: 2019-01-07.

[24] Mark Santolucito, Ennan Zhai, Rahul Dhodapkar, Aaron Shim, and Ruzica Piskac. 2017. Synthesizing Configuration File Specifications with Association Rule Learning. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 64 (Oct. 2017), 20 pages. https://doi.org/10.1145/3133888

[25] M. Sayagh and A. E. Hassan. 2020. ConfigMiner: Identifying the Appropriate Configuration Options for Config-related User Questions by Mining Online Forums. *IEEE Transactions on Software Engineering* (2020), 1–1. https://doi.org/10.1109/TSE.2020.2973997

[26] Patrick Stöckle, Bernd Grobauer, and Alexander Pretschner. 2020. *Repository to demonstrate the steps of the automated hardening process.* https://github.com/tum-i22/disa-windows-server-2016 swh:1:dir:c3803619f51702199b19405547e2be2f2f55bdd2.

[27] Patrick Stöckle, Bernd Grobauer, and Alexander Pretschner. 2020. *Repository with the check results for CIS guides before and after implementing the guides.* https://github.com/tum-i22/CIS-Benchmark-Evaluation swh:1:dir:b5c15f48b2c288f58533c9354bea3703ffbbb0dd.

[28] Patrick Stöckle, Bernd Grobauer, and Alexander Pretschner. 2020. Updated version of the step repository with Windows Server 2019. https://github.com/tum-i22/disa-windows-server-2019 swh:1:dir:13ffd9d2566c64afdedd414336a95a35605392d7.

[29] Ya-Yunn Su, Mona Attariyan, and Jason Flinn. 2007. AutoBash: Improving Configuration Management with Operating System Causality Analysis. *SIGOPS Oper. Syst. Rev.* 41, 6 (Oct. 2007), 237–250. https://doi.org/10.1145/1323293.1294284

[30] Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. 2007. /*Icomment: Bugs or Bad Comments?*/. *SIGOPS Oper. Syst. Rev.* 41, 6 (Oct. 2007), 145–158. https://doi.org/10.1145/1323293.1294276

[31] Chunqiang Tang, Thawan Kooburat, Pradeep Venkatachalam, Akshay Chander, Zhe Wen, Aravind Narayanan, Patrick Dowell, and Robert Karl. 2015. Holistic Configuration Management at Facebook. In *Proceedings of the 25th Symposium on Operating Systems Principles* (Monterey, California) *(SOSP '15)*. ACM, New York, NY, USA, 328–343. https://doi.org/10.1145/2815400.2815401

[32] David Waltermire and Jessica Fitzgerald-McKay. 2018. *Transitioning to the Security Content Automation Protocol (SCAP) Version 2.* Technical Report. NIST. Available from https://csrc.nist.gov/publications/detail/white-paper/2018/09/10/transitioning-to-scap-version-2/final.

[33] Rui Wang, XiaoFeng Wang, Kehuan Zhang, and Zhuowei Li. 2008. Towards Automatic Reverse Engineering of Software Security Configurations. In *Proceedings of the 15th ACM Conference on Computer and Communications Security* (Alexandria, Virginia, USA) *(CCS '08)*. ACM, New York, NY, USA, 245–256. https://doi.org/10.1145/1455770.1455802

[34] Edmund Wong, Lei Zhang, Song Wang, Taiyue Liu, and Lin Tan. 2015. DASE: Document-assisted Symbolic Execution for Improving Automated Software Testing. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1* (Florence, Italy) *(ICSE '15)*. IEEE Press, Piscataway, NJ, USA, 620–631. http://dl.acm.org/citation.cfm?id=2818754.2818831

[35] Tianyin Xu, Long Jin, Xuepeng Fan, Yuanyuan Zhou, Shankar Pasupathy, and Rukma Talwadker. 2015. Hey, You Have Given Me Too Many Knobs!: Understanding and Dealing with Over-designed Configuration in System Software. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (Bergamo, Italy) *(ESEC/FSE 2015)*. ACM, New York, NY, USA, 307–319. https://doi.org/10.1145/2786805.2786852

[36] Tianyin Xu, Xinxin Jin, Peng Huang, Yuanyuan Zhou, Shan Lu, Long Jin, and Shankar Pasupathy. 2016. Early Detection of Configuration Errors to Reduce Failure Damage. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 619–634. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/xu

[37] Tianyin Xu, Jiaqi Zhang, Peng Huang, Jing Zheng, Tianwei Sheng, Ding Yuan, Yuanyuan Zhou, and Shankar Pasupathy. 2013. Do Not Blame Users for Misconfigurations. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. ACM, New York, NY, USA, 244–259. https://doi.org/10.1145/2517349.2522727

[38] Tianyin Xu and Yuanyuan Zhou. 2015. Systems Approaches to Tackling Configuration Errors: A Survey. *ACM Comput. Surv.* 47, 4, Article 70 (July 2015), 41 pages. https://doi.org/10.1145/2791577

[39] J. Yang, E. Wittern, A. T. T. Ying, J. Dolby, and L. Tan. 2018. Towards Extracting Web API Specifications from Documentation. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*. 454–464.

[40] Zuoning Yin, Xiao Ma, Jing Zheng, Yuanyuan Zhou, Lakshmi N. Bairavasundaram, and Shankar Pasupathy. 2011. An Empirical Study on Configuration Errors in Commercial and Open Source Systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (Cascais, Portugal) *(SOSP '11)*. ACM, New York, NY, USA, 159–172. https://doi.org/10.1145/2043556.2043572

[41] Hao Zhong, Lu Zhang, Tao Xie, and Hong Mei. 2011. Inferring specifications for resources from natural language API documentation. *Automated Software Engineering* 18, 3 (01 Dec 2011), 227–261. https://doi.org/10.1007/s10515-011-0082-3