Submitted Version of: Hardening with Scapolite: a DevOps-based Approach for Improved Authoring and Testing of Security-Configuration Guides in Large-Scale Organizations

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

Ionuț Pruteanu
ionut.pruteanu@siemens.com
Siemens AG
Bucharest, Romania

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

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

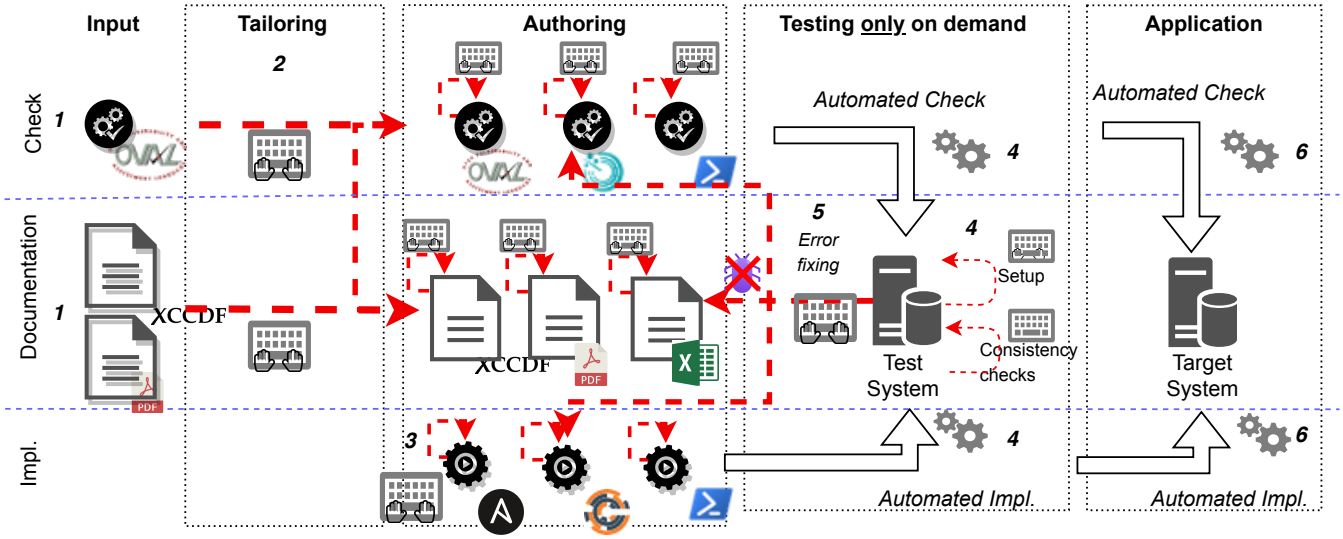

Figure 1: Typical process of security hardening. Dotted arrows represent manual tasks. Every arrow within the box is a task the administrators execute to harden the system.

ABSTRACT

FULLPAPER[1] Security Hardening is the process of configuring IT systems to ensure the security of the systems' components and data they process or store. In many cases, so-called security-configuration guides are used as a basis for security hardening. These guides describe secure configuration settings for components such as operating systems and standard applications. Rigorous testing of security-configuration guides and automated mechanisms for their implementation and validation are necessary since erroneous implementations or checks of hardening guides may severely impact systems' security and functionality. At Siemens, centrally maintained security-configuration guides carry machine-readable information specifying both the implementation and validation of each required configuration step. The guides are maintained within *git* repositories; automated pipelines generate the artifacts for implementation and checking, e.g., PowerShell scripts for Windows, and carry out testing of these artifacts on AWS images. This paper describes our experiences with our DevOps-inspired approach for authoring, maintaining, and testing security-configuration guides. We want to share these experiences to help other organizations

[1]We submitted this article as a full-length paper. Unfortunately, the CODASPY Program Committee decided that our paper can only be accepted in the tool track. Thus, the published version only consists of 6 pages.

CODASPY '22, April 24–27, 2022, Baltimore, MD, USA

ACM ISBN 978-1-4503-9220-4/22/04…$15.00
https://doi.org/10.1145/3508398.3511525

with their security hardening and, thus, increase their systems' security.

CCS CONCEPTS

• **Security and privacy** → *Usability in security and privacy*; *Software security engineering*.

ACM Reference Format:
Patrick Stöckle, Ionuț Pruteanu, Bernd Grobauer, and Alexander Pretschner. 2022. Submitted Version of: Hardening with Scapolite: a DevOps-based Approach for Improved Authoring and Testing of Security-Configuration Guides in Large-Scale Organizations. In *Proceedings of the Twelveth ACM Conference on Data and Application Security and Privacy (CODASPY '22), April 24–27, 2022, Baltimore, MD, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3508398.3511525

1 INTRODUCTION

Insecure configurations of operating systems and applications are known to be both common and detrimental to cybersecurity [5, 7, 9, 28, 35]. Organizations, therefore, need to identify the security-relevant configuration settings of the used software, determine the secure value or a set of secure values for each setting, and ensure that they configure each instance of the software used within their organization accordingly. This process is called security-configuration hardening and is part of the general security hardening of an organization's infrastructure. Security hardening is a continuous process rather than a one-time-only task since the IT infrastructure, the threat environment, insights about (in)secure configurations, et cetera are constantly in flux.

Organizations such as the Center for Internet Security (CIS) or the Defense Information Systems Agency (DISA) provide publicly available security-configuration guides (also called benchmarks, guidelines, or baselines) for various software components, e.g., operating systems like Windows 10, web servers like NGINX, or email clients like Outlook. These guides consist of rules, and each rule states which values should be used for a configuration setting relevant for security; some of these guides consist of more than 350 rules. Benchmarks written in the SCAP [30] standard often contain machine-readable definitions of *checks*, whereas mechanisms for *implementing* the required settings are usually either provided separately or not at all. The usual security-configuration hardening process, which is based on such public guides, contains many manual steps that are both inefficient and error-prone. Most of the time, we need to adapt the external guides for our target infrastructure by modifying specific settings, removing some rules, and adding others. This problem is intensified by the fact that these adaptions have to be replicated and kept consistent for each implementation, such as scripts (e.g., Bash or PowerShell), Infrastructure as Code (IaC) approaches (e.g., Ansible or Chef), et cetera, and for each check mechanism.

1.1 Problems of the Current Security Hardening Process

Figure 1 illustrates the usual security hardening process; the numbers in the figure refer to the following steps:

(1) Input is an external guide, usually in the SCAP standard: The human-readable parts are defined in the *eXtensible Configuration Checklist Format* (XCCDF) with machine-readable checks in the *Open Vulnerability and Assessment Language* (OVAL).
(2) XCCDF offers a mechanism for tailoring the guide, e.g., configure changes via so-called profiles. The profiles are also reflected in the OVAL-based checks.
(3) Because machine-readable implementation mechanisms are not part of these guides (exception: ComplianceAsCode, discussed below), we must either manually develop implementation mechanisms or adjust them if we can re-use existing mechanisms. Since larger organizations may use several different implementation mechanisms, we may need to re-apply the same changes numerous times.
(4) Before applying the implementation mechanisms to and using the check mechanisms for production systems, we must test both of them: Erroneous implementation and checking of security configurations may severely impact the security and functionality of systems. Because security-configuration guides are used for many target systems (different operating systems and applications, different releases, different tailorings, et cetera), we must manage a corresponding multitude of test systems.
(5) Feedback about problems, e.g., faulty implementations or checks, might introduce changes for one or several implementation/check mechanisms.
(6) Finally, the tailored and tested security guides can be applied to production systems. If problems are detected in productive use or a new version of a guide is published, the whole process restarts.

The repetition of these **manual** steps increases the risk of introducing **errors** and, thus, the risk of **insecure systems**. Therefore, we identified the following *challenges* for improving the security hardening:

- Remove superfluous complexity in the security hardening process resulting from unnecessary manual steps and scattered information.
- Establish automatic quality assurance for the security-configuration guides to find errors earlier and easier.

1.2 Our Approach: Improved Authoring, Artifact Generation, and Automated Testing

Our solution to these challenges is twofold. First, we present our improved configuration hardening approach that focuses on automation to remove error-prone manual steps. Second, we present our approach on automatic testing of security-configuration guides to detect errors as soon as possible.

Figure 2 shows our improved security hardening process; again, the numbers refer to the steps below:

(1) We manage security-configuration guides in a dedicated YAML-based format called *Scapolite*, which we keep under version control. Further, we enrich the format with machine-readable information about configuration requirements. Ideally, both implementation and check mechanisms can be automatically derived. Thus, we keep information about the

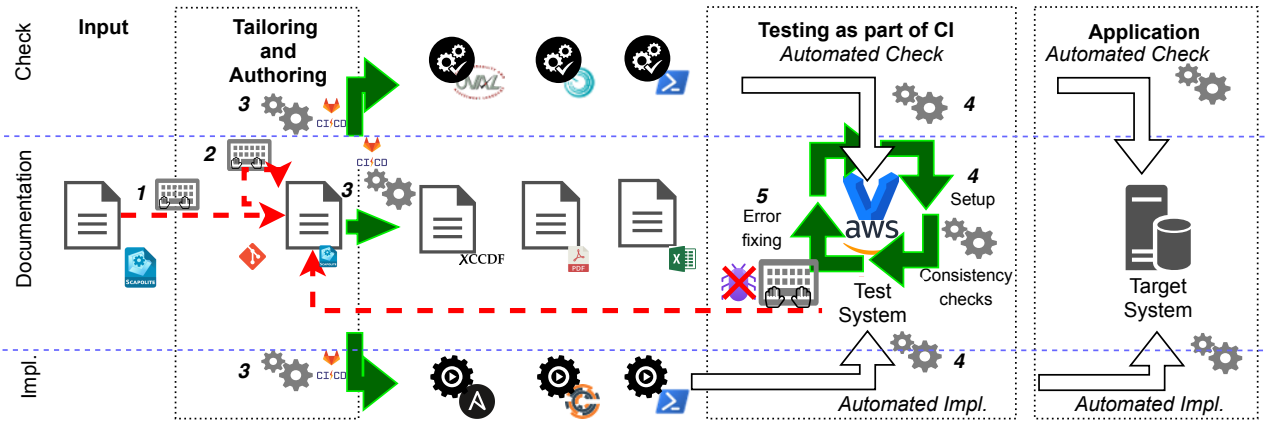

Figure 2: Improved process of security hardening. The green arrows represent activities that have been automated.

check, implementation, metadata, and documentation, e.g., human-readable descriptions about the requirements, the rationale, et cetera, at a single location.[2]

(2) Tailoring to different use-cases in Scapolite works similarly as in SCAP: We can define profiles for the individual use cases and create per-use-case modifications.

(3) From this single source, i.e., the machine-readable information from 1), we automatically generate the required artifacts for implementing/checking the guides.

(4) Creation of the required test systems as virtual machines, applying the implementations/checks to these systems, and collecting the test results is carried out automatically as a part of a DevOps pipeline.

(5) Because the implementations/checks are generated automatically, we can fix detected problems with a single change either in the Scapolite document defining the guide or a bug-fix in the transformation system, rather than changing in several different artifacts.

1.3 Contributions

Our contributions to the field of security hardening are:

- By pulling information required for generating both implementation and check mechanisms as machine-readable information into our security-configuration guides, we manage to restrict manual changes/corrections to a single location, thus reducing errors and increasing efficiency.
- We show how to operate a DevOps/Continuous Integration-inspired approach of authoring and maintaining security-configuration guides. In our approach, changes in the guides trigger automated tests without human involvement in the execution of the tests, collection of test results, and correlation of test data with expected results.

[2]External guides in SCAP can be automatically converted into Scapolite. Adding machine-readable information from which implementations and checks can be derived requires, of course, manual effort, but such effort would be necessary for generating separate implementation mechanism, as well. Furthermore, for some use-cases, semi-automated mechanisms for deriving machine-readable information from human-readable text exist [26].

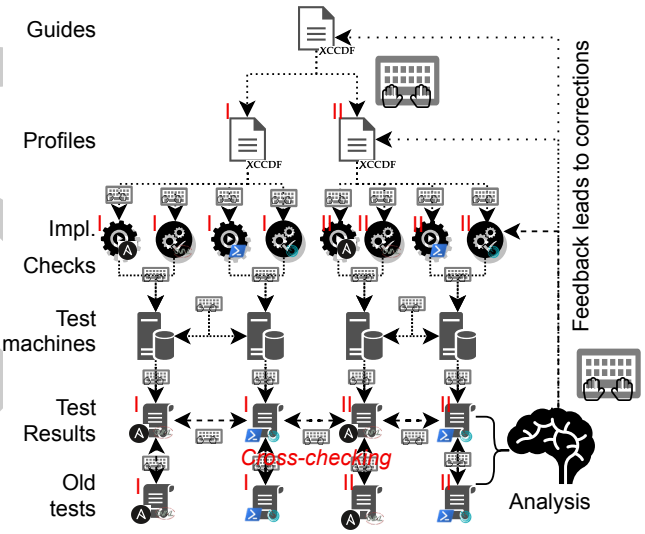

Figure 3: Regular execution of tests in a security hardening process. Dotted lines denote manual tasks. Every arrow within the box is a task the administrators execute to harden the system.

The latter point deserves a closer examination: As explained above, security-configuration mechanisms are affected by the combinatorial explosion of test cases, requiring many test systems and test runs. Figure 3 illustrates the approach without the DevOps: a single test already requires a substantial manual effort that must be multiplied by the number of test systems/test cases; when we detect problems, we have to fix them at several locations. In contrast, Figure 4 illustrates the level of automation of our approach.

Our experiences of handling multiple security-configuration guides with multiple profiles authored/maintained using version control and DevOps pipelines within an industrial context show

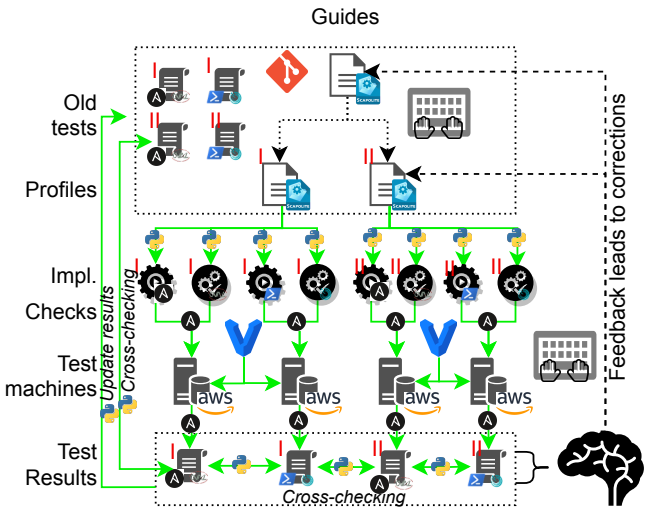

Figure 4: State-of-the-art execution of tests in a security hardening process. The green arrows denote steps that are now automated.

that an approach that combines machine-readable information required for implementing and checking security-configuration requirements is not only feasible but provides enormous benefits. Errors are reduced, and the efficiency and the effectiveness of an organization's security-configuration hardening program are raised. Thus, we tackle two of the major causes for insecure configurations: erroneous application and ineffective or incomplete application of secure configurations.

2 OUR APPROACH TO SECURITY HARDENING

Everyone who has published security-configuration guidelines to their organization, e.g., a document specifying the required security settings for a *Windows* or *Linux* server system, will be familiar with the demand for means that allow automated implementation and validation of these settings. Especially in the case of operating systems, for which the number of relevant settings is over 350, publishing a guide without providing automated mechanisms is both inefficient and ineffective:

- multiple persons/groups in the constituency work in parallel on creating implementation/validation mechanisms;
- the manual transcription of required settings into an implementation mechanism or a fully manual implementation will lead to errors and omissions;
- some constituency members will deem the task of implementation as too arduous, costly, or time-consuming and not bother with it at all.

The SCAP [30] format family defines the state of the art for providing automated mechanisms along with a security-configuration guide. We can use the SCAP formats to augment human-readable information with machine-readable checks, usually specified in OVAL [16]. In almost all cases, however, automated implementation mechanisms are maintained separately: both CIS and DISA provide Windows backup files containing the required settings, which need to be maintained manually – a cumbersome and error-prone process, as outlined above. The notable exception is the *ComplianceAsCode* [3] project that provides little scripts or Ansible playbooks for many settings. ComplianceAsCode includes the scripts in the resulting SCAP content such that tools can use them to carry out the implementation steps. At Siemens, we take a similar approach to ComplianceAsCode. However, we try to operate at a higher level of abstraction – where possible – by specifying the desired configurations in a machine-readable form such that we can derive both implementation and verification mechanisms from it. We combine this with a rigorous "DevOps"-approach for authoring and maintaining security guides: we use DevOps pipelines for both automated derivation and test of implementation and validation mechanisms. In the following, we will briefly outline our approach towards the abstract specification of security-configuration requirements and their automated transformation into implementations and checks.

2.1 The Scapolite Format

The starting point of our work was the definition of a format called "Scapolite," which encompasses the relevant features of SCAP but additionally provides

(1) a form that can be created/maintained as text-files under version control (cf. above comment on changes in rules).
(2) generalizations and additional extension points to support a broader range of use cases.
(3) fields for tracking of document maintenance data such as change history information per configuration requirement.

Similar to other projects [15, 21] that require a "human read- and writable" format for creating and maintaining structured information, we chose YAML[4] as a basis for Scapolite. Further, we combined YAML with Markdown[5] as a markup language for structuring human-readable content. We do not argue that SCAP and its XML formats OVAL, XCCDF, et cetera are not human-readable, but our experience from working with guide authors at Siemens shows that they are more motivated to write guides in a YAML/Markdown than in an XML format.

Listing 1 shows a minimal example of Scapolite; the highlighted lines contain the human-readable description of how to implement the required setting.

2.2 Adding Machine-Readable Automations

The setting prescribed by the rule in Listing 1 concerns a Windows policy setting, specified via (1) a policy path and (2) the required *policy value*. We, therefore, augment the Scapolite rule object shown in listing 1 with a so-called *automation* structure: the `implementation` section of that object has an optional keyword `automations` under which we can add a list of such *automation* structures. Listing 2 shows the required automation structure for this particular rule. Line 2 contains the policy path; starting with line 4, one can see

[3] https://github.com/ComplianceAsCode/content
[4] https://yaml.org/
[5] https://daringfireball.net/projects/markdown/

```
---
scapolite:
    class: rule
    version: '0.51'
id: BL942-1101
id_namespace: org.scapolite.example
title: Configure use of passwords for removable data drives
rule: <see below>
implementations:
  - relative_id: '01'
    description: <see below>
history:
  - version: '1.0'
    action: created
    description: Added so as to mitigate risk SR-2018-0144.
---
## /rule
Enable the setting 'Configure use of passwords for removable
data drives' and set the options as follows:
   * Select `Require password complexity`
   * Set the option 'Minimum password length for removable data drive` to `15`.
## /implementations/0/description
To set the protection level to the desired state, enable the policy
`Computer Configuration\...\Configure use of passwords for removable data drives`
and set the options as specified above in the rule.
```

Listing 1: A very basic example of a rule in Scapolite. Lines referenced in the text are marked in blue. We shortened the policy path to keep the file concise.

the required values. In addition to the policy path and the values, in lines 7-10, we also specify constraints for compliance checking: obviously, a password length > 15 would also be compliant.

```
system: org.scapolite.implementation.win_gpo
ui_path: Computer Configuration\...\Configure use of passwords for removable
↪ data drives
value:
  main_setting: Enabled
  Configure password complexity for removable data drives: Require password
  ↪ complexity
  Minimum password length for removable data drive: 15
  constraints:
    Minimum password length for removable data drive:
      min: 15
```

Listing 2: Windows-policy automation specifying a policy path, value(s) and constraints for compliance checking

We can configure Windows policies via a GUI interface, which allows the user to choose the desired values for each existing policy path. For a programmatic implementation, however, an intermediate step is necessary. In the case of this particular policy, we must set a specific key-value pair in the registry. We have, therefore, implemented an automated transformation of the policy-based specification to a registry-based automation (similar transformations exist for other "low-level" mechanisms required for other Windows policies).

2.3 Transforming Automations

Listing 3 provides the result of carrying out this transformation for the automation in Listing 2: we must set three registry keys; the first key signifies that the setting is enabled; the second specifies that the requirements on password complexity are active; the third contains the minimum password length.

Ideally, all security requirements should be specified as abstractly as possible and then be transformed automatically into mechanisms for implementation and checking. However, if we cannot find a suitable abstraction level, we must include code in a suitable scripting language. For expressing checks, we can at least regain some abstraction via a generic method for expressing the expected output of check-scripts to keep the scripts included as "script automation" in the Scapolite document as concise as possible. Listing 4 shows an example of a check for the requirement that all mounted volumes larger than 1GB should use the NTFS file system. Lines 6-8 specify the expected output: the script in line 3 returns a list of information objects, each of which must carry the key-value pair `FileSystemType:NTFS`.

```
system: org.scapolite.automation.compound
automations:
  - system: org.scapolite.implementation.windows_registry
    config: Computer
    registry_key: Software\Policies\Microsoft\FVE
    value_name: RDVPassphrase
    action: DWORD:1
  - system: org.scapolite.implementation.windows_registry
    config: Computer
    registry_key: Software\Policies\Microsoft\FVE
    value_name: RDVPassphraseComplexity
    action: DWORD:1
  - system: org.scapolite.implementation.windows_registry
    config: Computer
    registry_key: Software\Policies\Microsoft\FVE
    value_name: RDVPassphraseLength
    action: DWORD:15
    constraints:
        min: 15
```

Listing 3: Example of the Windows Registry automations generated from Listing 2

```
system: org.scapolite.automation.script
script: |
    Get-Volume | Select Size, FileSystemType | Where {$_.Size -gt 1GB}
expected:
    output_processor: Format-List
    each_item:
      key: FileSystemType
      equal_to: NTFS
```

Listing 4: Example of a script-based automation for checking that all drives larger than 1GB use NTFS as their file system type.

2.4 Producing Code and Other Artifacts

With (1) the machine-readable specifications of what needs to be implemented/checked and (2) the associated transformation mechanisms, we can generate artifacts that the system administrators can use to carry out the rule's implementation and check. The higher our level of abstraction, the more options we have regarding the target implementation or check mechanism for which we generate these artifacts. Obviously, if the automations contain code for a specific script engine, we must generate artifacts for each of these engines or an execution system that can execute this type of script.

For this article, we continue the example regarding Windows. For security-configuration guides targeting Windows, we generate a set of PowerShell commandlets together with a JSON file containing for each rule the necessary data used by the PowerShell commandlets to implement or check the rule. Before the scripts implement a rule, they store as backup each setting's current value; Thus, we can roll back every implemented rule.

As an example for a different *target* of our transformations, Listing 5 shows the result of a transformation from Listing 3 into an OVAL check. This particular transformation might look straightforward, but even simple checks can get complicated when expressed in OVAL; combined with the verbose XML structure of OVAL and its many cross-references, generating OVAL was a prime use case for our code generation.

Our improved approach to security hardening has several advantages: First, it concentrates all information of a single rule in one place and reduces the risk of inconsistencies. Second, the transformations replace many manual steps and thus significantly reduce the risk of errors.

```
<criteria negate="false" operator="AND">
  <criteria negate="false" operator="AND">
    <criterion negate="false" test_ref="oval:tst:105650">
      <win:registry_test check="all" check_existence="at_least_one_exists"
      ↪ id="oval:tst:105650" version="1">
        <win:registry_object id="oval:obj:105650" version="1">
          <win:hive datatype="string" operation="equals">
            HKEY_LOCAL_MACHINE
          </win:hive>
          <win:key datatype="string" operation="case insensitive equals">
            Software\Policies\Microsoft\FVE
          </win:key>
          <win:name datatype="string" operation="equals">
            RDVPassphrase
          </win:name>
        </win:registry_object>
        <win:registry_state id="oval:ste:105650" version="1">
          <win:type datatype="string" operation="equals">
            reg_dword
          </win:type>
          <win:value datatype="int" entity_check="all" operation="equals">
            1
          </win:value>
        </win:registry_state>
      </win:registry_test>
    </criterion>
  </criteria>
  ...
</criteria>
```

Listing 5: Parts of an OVAL check (nested for better readability) generated from listing 3. Shown is the part of the check that considers the first of the three registry keys.

3 THE NEED FOR AUTOMATED TESTING

Having explained how we specify security-configuration requirements and transform these specifications into artefacts for implementation and checking, we move to motivating the need for extensive test automation as part of our maintenance and release process in the following section.

3.1 Maintenance and Release Process

Our workflow in authoring, maintaining, and releasing security-configuration baselines is as follows:

(1) Authors write security-configuration guides using Scapolite. The Scapolite files are kept under version control at `code.siemens.com`, an internal *GitLab* instance.
(2) We use GitLab pipelines to automatically transform the machine-readable automations into artifacts for implementation and check, i.e., in the Windows case, we generate JSON files and PowerShell scripts. During the development or maintenance of a guide, the authors use these guides for testing purposes.
(3) Once we release a guide, Siemens's security-regulation portal called *SFeRA* generates human-readable versions (web view, PDF, XLSX, etc.) directly from the Scapolite sources located at `code.siemens.com`.
(4) The pipeline-based transformation mechanism is triggered for the released version of the Scapolite sources, and we provide the resulting artifacts to users via dedicated GitLab repositories.

In a parallel process, we maintain the technological basis of this process and develop it further, namely:

(1) libraries for creating and manipulating Scapolite content, e.g., imports from SCAP, methods for enriching existing Scapolite content with additional information, et cetera;
(2) libraries for transforming abstract machine-readable automations into more concrete automations, e.g., transforming a Windows policy requirement into registry key settings (cf. Section 2.3);
(3) libraries for further transformation into code or other artifacts (cf. Section 2.4)

3.2 Combinatorial Explosion of Needed Test Cases

Before describing the test requirements during the creation/maintenance of security-configuration guides, it is worthwhile to consider the number of test cases for a given guide.

Usually, we write security-configuration guides to serve different use-cases with the same guide. We normally specify different security levels, where specific rules only apply to particular levels or rules are modified according to the security level. For example, a lower password length may be required for standard systems, whereas we specify a longer password length for high-security systems or add a rule mandating two-factor authentication.

Also, frequently, we differentiate between other use-case variants such as client and server systems. Thus, a scheme for defining system criticality or sensitivity in three levels for an OS, i.e., low, medium, high, as well as for two roles, i.e., client and server, will lead to 6 test cases. For Siemens's Enterprise IT, we use a criticality schemes which (in theory) can lead to 27 different possible criticality levels.

Finally, a single security-configuration guide may apply to several releases of its target, e.g., Windows releases (1809, et cetera), or different editions or flavors of the target, e.g., CentOS vs. RHEL.

Thus, we see that testing of security guides suffers from a substantial combinatorial explosion problem. We know mitigation strategies, e.g., containerization, to provide a controlled execution environment to remove the variability; we cannot apply them since security-configuration guides strive to be applicable as widely as possible.

3.3 Test Requirements during Guide Creation

Creating a security-configuration guide is an iterative process between writing the guide and testing the guide's implementation. The author, therefore, requires a test environment, usually in the form of one or more virtual images on which the target of the baseline is installed.

Manual creation/maintenance of such a test environment, as well as the manual execution of the tests, is a tremendous overhead: we must start/reset the virtual image, generate the artifacts, transfer them to the image, and execute the artifacts; usually, we execute this process several times for implementing and checking rules for different use-cases. In the end, we must collect the test results and prepare them for the manual analysis.

The efficient creation of security-configuration guides, therefore, is impossible without automated testing.

3.4 Test Requirements During Guide Maintenance

Automated testing also is essential during maintenance. Every change either in the Scapolite source or the underlying infrastructure required for generating the artifacts for implementation and checking may lead to errors. For example:

(1) Errors in the metadata introduced during maintenance may lead to rule omissions in the generated artifacts.
(2) Errors in the transformation from abstract to concrete machine-readable information may lead to faulty specifications, which in turn lead to faulty implementations and checks. These transformation errors can originate from, e.g., bugs introduced during maintenance of the transformation library.
(3) Similarly, errors in the transformation to program code or other artifacts may lead to faulty implementations/checks.

Further, we need to detect errors in a timely manner that are introduced by changes that have nothing to do with our process:

(1) Maintainers may misspecify the machine-readable information when making changes during maintenance.
(2) Changes in the target of hardening, e.g., upgrades of the OS, may invalidate or break a particular way of implementing or checking.
(3) Changes in execution environments for a created artifact, e.g., changes in a vulnerability scanner we generate a specification for, may invalidate the created artifact.

Only a high automation degree allows us to run the required regression tests whenever a change occurs.

4 OUR APPROACH TO TESTING

4.1 The Testing process

As pointed out in Section 3.2, testing the implementation and checking of a security guide to a target is likely to require several test runs: one for each combination of use-case, e.g., regular vs. high-security, used system, target-system revision, e.g., Windows release 1809 vs. 1909, and implementation or check runtime environments; the latter either ingest some of the created artifacts, e.g., a test policy, or provide as external mechanisms a certain *ground truth*. We use, for example, the CIS-CAT scanner to verify implementations/checks generated for CIS baselines. Nevertheless, we can also have different results for the same tools, e.g., because of different versions.

4.1.1 Anatomy of typical test run. A test run typically has the following shape:

Run initial checks Run checks on the unchanged system to establish the status quo *before* the implementation.
Apply security settings Execute the generated mechanism for implementing the desired security settings.
Carry out checks for compliance Re-run checks against the changed system.
Revert settings Revert the revertable settings to their initial status.
Check reverted settings Check the status after we restored the settings' old state.

4.1.2 Analysis of a test run. Relevant data that can be collected from such test runs are:

Quantitative data How many rules were successfully applied? For how many rules did the check return a success, a failure, a runtime problem, et cetera?
Detailed information Which rules were successfully applied? For which rules was the check successful, a failure, ran into a problem, et cetera?

Analysis of the complete set of test runs for a specific setting, i.e., a combination of use-case and target system, usually entails two types of comparison:

Comparisons within a test run to find discrepancies, e.g.:
- A rule is reported as applied, but the check mechanism reports the rule as non-compliant.
- Two check mechanisms report different results for a rule.
- The check mechanism marked a rule as non-compliant before the implementation, compliant after the implementation, but still as compliant after the reverting.

Comparison with previous test runs to carry out regression tests: the newly collected data is compared with data from previous test executions. Were there changes? If so, are these *desirable* changes, e.g., we improved an implementation or check that did not work before, or *undesireable* changes, e.g., previously successful check does not succeed anymore.

4.2 Our Approach to Test Automation

In order to automate testing as much as possible, we implemented the following approach: Our tooling automatically executes a machine-readable test specification on VMs created on-demand in AWS; the tooling carries out the specified test activities, collects the raw data generated from implementation and check mechanisms, and automatically prepares summary data and data comparisons required to analyze the tests.

This complete automation of test activities allows an author or maintainer to carry out tests with no effort; the extensive preprocessing of the test data enables them to see directly whether there are deviations from the expected results and enables them to focus on analyzing the cause of these deviations.

4.2.1 Test Specification. With our YAML-based file format, we can define one or more test runs; they are executed on different instances in parallel. We specify:

- for each test run, a sequence of activities such as implementing, checking, or reverting rules (cf. Section 4.1.1);
- for each activity, a list of so-called validations; each validation compiles data from the result or log files created by an activity (for

example, validations can count successfully checked rules, collect these rules' identifiers, compare the current results to results of previous activities, et cetera);
- for each validation, the expected results (as basis for regression tests along with each validation)

The test specification file is kept under version control with the Scapolite sources for each security-configuration guide.

4.2.2 Test Execution. We have implemented a test runner that is part of the DevOps pipeline that generates the artifacts for implementation and checks. The test runner accesses the test specification file in the repository and executes the tests:

- For each test run, the runner starts the required AWS image.
- The runner transfers the created artifacts and additional resources required for implementation/checking to the image.
- The runner uses Ansible to carry out the specified activities.
- In the end, the runner retrieves the created result/log files from each activity from the image, stops and destroys it.

4.2.3 Preprocessing of test results. As described in Section 4.2.1, we can specify validation tasks for each action carried out in the test run. Hence, after the runner collected all raw data, the tooling carries out the validation tasks: the required data is compiled, and a comparison to the expected results specified in the test specification file is carried out.

As a final step, our tooling commits (1) a detailed log, (2) a report of found deviations, (3) an updated test specification file with the current validation results, and (4) all raw data retrieved from the image to a staging repository.

4.3 Test Specification

4.3.1 Structure of the test file. Listing 6 shows an exemplary test specification file. As detailed in Section 4.2.1, each test run specifies several activities with a list of validations per activity (colored lines are referred to below):

- We specify two test runs (lines 5-6), one for the *Level 2*, i.e., high-security, profile of a CIS Windows 10 (1809) Benchmark, the other one for the basic *Level 1* profile. Here we only show parts of the latter.
- As explained in Section 4.1.1, we start with a check of the unchanged system, using the generated PowerShell scripts (line 11). The first validation activity (lines 15–21) provides a count of the check result: how many rules were compliant, non-compliant, et cetera. Here, as in all the following examples, the values defined in the test specification file are the expected values taken from previous test runs.
- We continue using the generated PowerShell scripts to apply all rules (line 25) of the chosen *Level 1* profile (line 7). As we will discuss in more detail in Section 4.4.3, we usually need to blacklist some rules (line 26) because there are rules breaking the test mechanism, e.g., by disrupting connections to the test machine. Again, amongst other things, we validate the number of successfully applied rules (line 30).
- We follow the rules' application with two check activities: we check with the generated PowerShell script (lines 33ff) and an external scanner provided by the CIS [3] (lines 42ff).

```
os_image: Windows10                                                                    1
os_image_version: 1809                                                                 2
ciscat_version: v4.0.20                                                                3
testruns:                                                                              4
- name: 1809 L2 High Security (...)                                                    5
- name: 1809_Level1_Corporate_General_use                                              6
  testrun_ps_profile: L1_Corp_Env_genUse                                               7
  testrun_ciscat_profile: cisbenchmarks_profile_L1_Corp_Env_genUse                     8
  testrun_benchmark_filename: CIS_Win_10_1809-xccdf.xml                                9
  activities:                                                                          10
  - id: initial_powershell_check                                                       11
    type: ps_scripts                                                                   12
    sub_type: check_all                                                                13
    validations:                                                                       14
    - sub_type: count                                                                  15
      expected:                                                                        16
        blacklist_rules: 0                                                             17
        compliant_checks: 75                                                           18
        non_compliant_checks: 272                                                      19
        empty_checks: 2                                                                20
        unknown_checks: 2                                                              21
    (...)                                                                              22
  - id: apply_all                                                                      23
    type: ps_scripts                                                                   24
    sub_type: apply_all                                                                25
    blacklist_rules: [R2_2_16, R2_3_1_1, ..., R18_9_97_2_4]                            26
    validations:                                                                       27
    - sub_type: count                                                                  28
      expected:                                                                        29
        applied_automations: 336                                                       30
        not_applied_automations: 4                                                     31
    (...)                                                                              32
  - id: check-after-apply-all-with-ps                                                  33
    type: ps_scripts                                                                   34
    sub_type: check_all                                                                35
    validations:                                                                       36
    - sub_type: by_id                                                                  37
      result: non_compliant_checks                                                     38
      comment: Correspond to blacklisted rules                                         39
      check_ids: [R2_2_16, R2_3_1_1, ..., R18_9_97_2_4]                                40
    (...)                                                                              41
  - id: check_after_apply_all_ciscat ...                                               42
    type: ciscat                                                                       43
    validations:                                                                       44
    - sub_type: compare                                                                45
      compare_with: check-after-apply-all-with-ps                                      46
      expected:                                                                        47
        comment: CISCAT error for 18.8.21.5                                            48
        rules_failed_only_here: [R18_8_21_5, ...]                                      49
        rules_unknown_only_here: [R1_1_5, R1_1_6, R2_3_10_1]                           50
        rules_unknown_only_there: [R18_2_1, ...]                                       51
        rules_passed_only_here: []                                                     52
    (...)                                                                              53
static:                                                                                54
- id: validate_json_file                                                               55
  type: examine_sfera_automation_json                                                  56
  validations:                                                                         57
  - sub_type: count                                                                    58
    expected:                                                                          59
      no_automation: 1                                                                 60
      (...)                                                                            61
  - sub_type: by_id                                                                    62
    expected:                                                                          63
      no_automation: [R18_2_1]                                                         64
      same_setting: []                                                                 65
(...)                                                                                  66
```

Listing 6: A summarized version of a test specification file.

  - Here, we see an example of validating not just rule counts but the actual rule identifiers, e.g., as we examine the rules that our script reports as non-compliant (line 40). In line 39, a tester made a comment: the non-compliant rules correspond to the blacklisted rules (in line 26).
  - We can also carry out other relevant comparisons automatically: For example, in lines 45ff., the check results of the CIS scanner are compared with the results of our PowerShell script; in line 49, under the keyword `rules_failed_only_here`, we see a list of rules which the CIS scanner reports as non-compliant, but our

PowerShell scripts report as compliant. Again, a tester added a comment (line 48) about the reasons for the deviations.
For example, for a specific rule, the CIS scanner requires that a particular setting should not be configured, even though the human-readable description of the rule requires that the setting should be disabled. Testers at Siemens re-discovered systematic false positives like these repeatedly; by documenting such problems of external scanners, testers can better focus on actual deviations.

- We also carry out static tests on the created artifacts (line 54ff.); the static tests are always carried out as the very first test activity. For example, we examine the created JSON file for entries without an automation (lines 60, 64) to catch errors during maintenance, leading to a failure when creating automations. Another valuable check is whether the same security setting is affected by several rules (line 65) since this often points to an error made during the rules' specification.

4.3.2 Management of the test specification file. When a test is carried out for the first time, the tester specifies the test runs, actions, and validations but leaves the fields about expected values empty since she does not know the expected values so far. When the test is completed, the test infrastructure generates a version of the test specification file that contains all values from the tests' results. The tester can use this version of the file as a basis for the following tests.

We manage the test specification file and the Scapolite sources that are the input to the pipeline in the same repository rather than at a separate location; similar to a `.gitlab-ci.yml`, we store the test specification file under `.scapolite_tests.yml`. Thus, during authoring/maintenance, when we create different branches, the test specification file is always part of the particular branch, drives the branch, and test results are fed back into the test specification file as expected results.

4.4 Execution of Tests

4.4.1 Testing in the cloud. Our test infrastructure started as a server equipped with VirtualBox[6] for creating test images; furthermore, we used Vagrant[7] to manage image creation and destruction, Ansible for carrying out the test activities, and transferring data between the server and the images.

This approach, though well-suited for developing the test infrastructure, could not scale. The combinatorial explosion in test cases that occurs for security-configuration guides often leads to many test cases. Thus, we firstly must run all test runs for a single test in parallel to keep the time for executing a complete test acceptable. Secondly, we need several authors/maintainers to work in parallel without the scarcity of test resources hindering them.

We, therefore, moved the testing process into the cloud and migrated from Oracle VirtualBox to AWS EC2[8]. In the beginning, we had to overcome some initial problems caused by differences between VirtualBox and EC2 in credential management and the access of virtual machines. Also, we had to redesign how we transfer data between the test runner and the images. Using VirtualBox, the transfer of big files, e.g., the CIS-CAT scanner and a JVM to run it on, is essentially a local file-copy operation, whereas, with EC2, a naïve implementation would constantly transfer these files via the Internet from the local test runner to EC2. We thus integrated an S3 bucket into our architecture, in which we host the files required for each test run: hence, we transfer the data rather within the AWS data center than via the internet.

4.4.2 Integration into DevOps pipeline. We generate the artifacts for implementing and checking security configurations from the Scapolite sources with a DevOps pipeline maintained as a GitLab-CI *include file* within a dedicated repository. For each Scapolite repository, we include this file into the GitLab-CI file; because we factored out the actual code for the pipeline, we (1) keep the project's CI file very concise with only project-specific definitions, and (2) can carry out the maintenance of the pipeline via the single pipeline repository.

In code development, when changes are pushed to the code repository, tests are run changes are run. In our case, however, each test entails the creation of several virtual machines, and the execution of a test run may take up to an hour. We, therefore, chose to carry out only static tests for each push but require an active request by the author/maintainer for dynamic tests; we realized this via a pipeline variable `EXECUTE_TESTS` passed to the pipeline.

4.4.3 Dealing with negative effects of secure configurations on test execution. In Section 4.3.1, we mentioned the `blacklist` definition required in test activities that implement security settings to preserve the test infrastructure's functionality. The infrastructure relies on specific mechanisms for accessing and manipulating the VM on which we carry out the tests. Usually, the guides recommend disabling some of these mechanisms, e.g., firewall rules, rules restricting the use of stored credentials, et cetera, which may disrupt the WinRM functionality that Ansible uses. If we implemented one of these rules, following test activities would cause Ansible to fail, with little or no information about why the activity failed. In order to help the users with finding rules that break the test infrastructure, we implemented the following features:

- Users can implement the rules in an *apply* activity one by one rather than in bulk. A failure in execution can thus usually be attributed to the rule applied just before the failure occurred.
- To speed up test execution in this process of finding rules to blacklist, they can configure the rule implementation to start either at a specific rule or at the last rule contained in the blacklist; the guide specifies the rules' order. Unless a combination of rules causes an execution failure, this suffices to find all rules that must be blacklisted.

4.5 User Feedback

As shown above, we have highly automated the testing process itself. However, the analysis of the test results still requires human interaction. It is thus necessary to present the test results such that they provide the user with a concise overview of whether something went wrong and allow easy access to the raw data necessary for an in-depth analysis of problems uncovered by the test.

4.5.1 Summary Report. Once we executed all test runs and the analyses and comparisons specified for each activity have been

[6] https://www.virtualbox.org/
[7] https://www.vagrantup.com/
[8] https://aws.amazon.com/ec2/

carried out, our tooling generates a summary report providing concise information for each activity:

(1) Did failures occur during an activity, e.g., because a setting interrupted the connection to the virtual image and the activity could not be completed?
(2) If no failure occurred, did the test yield the expected results as documented in the test specification file?
(3) Where possible: if the test yielded different results, did the test show an *improvement*? Were more rules implemented successfully than during the previous test run?

With item 3) we intend to provide the user with an initial assessment of the test results. This, however, requires a definition of what constitutes an improvement/degradation. The users can specify in the test specification file what an improvement should be along with the expected data. For example, the key-value pair `improvement:rise` in combination with a validation that counts results, e.g., the number of successfully implemented rules or of checks showing compliance, signifies that a reported higher number constitutes an improvement; `improvement:fall` would do the opposite. If no condition for an improvement is specified, a degradation is reported by default if the test results do not match the expected data.

4.5.2 Documentation of full results. In case a deeper analysis of the results becomes necessary, the users can access detailed information about found deviations for each validation step: Listing 7 provides an example of how a deviation is reported. Furthermore, users can access the raw data for each activity within a staging repository containing the generated artifacts. Thus, all relevant data are provided at one location. Also, they can use different mechanisms provided by *git* and *GitLab* such as viewing differences between test executions, e.g., within the generated artifacts, during the analysis of the test results.

```
1  CRITICAL - Validation failed, SAME numbers, but DIFFERENT IDs (IMPROVEMENT:
   ↪  'fall')!
2   Expected and confirmed(found) 'unknown_checks' IDs: {'R18_2_1', 'R2_3_1_6',
   ↪  'R2_2_21', 'R2_3_1_5'}
3   Expected 'unknown_checks' IDs, but not found: {'R2_3_11_3'}
4   Found 'unknown_checks' IDs, but not expected: {'R19_7_41_1'}
```

Listing 7: Example report of a difference between test results and expected results.

4.5.3 Further automation. We provide further support to the users if they need to re-test several guides, e.g., when the transformation mechanism was updated. These command-line scripts that use the *GitLab* API include tasks like:

- starting pipelines in parallel for several guides;
- informing about the pipelines' status;
- compiling an overview with the results of all test pipelines;
- showing differences between the newly-generated artifacts and the latest published version for each guide;

By automating repetitive manual tasks carried out for each guide, we achieve that tests are executed frequently. Especially small or seemingly *harmless* changes are now more often tested because we lowered the effort for starting the tests and analyzing the test results for more than one guide significantly.

5 RELATED WORK

First, we present the current work on configuration management in general and security hardening in particular. Second, we discuss approaches similar to our testing approach.

In past, reseachers investigated heavily in misconfiguration in general, and security misconfiguration in particular [5, 7, 9, 28, 35]. Dietrich et al. [7] show in their study that security misconfigurations are very common and a severe problem. According to their data, manual configuration, vague or no process, and poor internal documentation are the main environmental factors that we could solve with a better approach and tooling.

Many researchers investigated how we can detect and remove misconfigurations [11, 20, 24, 27]. Rahman et al. [20] analyzed thousands of IaC scripts to identify insecure configurations; the framework *ConfigV* [24] learns good configuration settings based on given configuration files. Depending on the guide's target, such techniques could be used to develop the guide or check for problems with the chosen configuration settings by applying them to the generated implementation artifacts. SPEX [34] examines the source code of programs in order to find security-related configurations and would thus be useful in the creation of public guides for open-source software as well as internal guides for one's products.

The creation of automated implementation/check mechanisms becomes much easier when a unified framework for setting and checking configurations for a software product is in place. The Elektra framework [19], for example, unifies how we can access configuration settings and creates a central structure for accessing and manipulated configuration settings. Xu et al. [33] developed a similar approach to Elektra. Furthermore, they showed [32] convincingly that the configuration's complexity is overwhelming users and systems administrators. The results of the study underline how important security experts and security guides are in supporting the administrators.

The ComplianceAsCode project [18, 22] is very close to the presented approach. The authors maintain their security-configuration guides for various Linux systems in a git repository and represent every rule with one file. This file references other files, e.g., with scripts for automated checking. Nevertheless, some drawbacks prevented us from using ComplianceAsCode. First, their focus on Linux-based operating systems did not support our initial, primary use of Windows hardening. Second, in contrast to ComplianceAsCode, we try to generate as much as possible from a single abstract specification, whereas ComplianceAsCode maintains a check in OVAL and the implementation mechanism(s) for each setting in a different language. Nonetheless, it would ease the security configuration enormously if the publishers distributed their guides in a format akin to ComplianceAsCode so that documentation, check *and* implementation are more aligned.

Software testing is a well-researched discipline, and every year, new articles add more information to the general knowledge [1, 2, 4, 6, 8, 12, 13, 17, 23, 29]. Therefore, we can only refer to a fraction of all available and valuable testing research. Many researchers, e.g., [1, 10] use sophisticated testing approaches to find security-relevant bugs or leaks in software. In contrast, we use testing approaches to find bugs in the security-configuration guides, not the software itself. In industry, there is a strong need for automated testing,

especially in the DevOps scenarios [14]. Also, there are some obstacles to overcome, e.g., when testing a software's graphical user interface [31]. Since we use our approach productively at Siemens, we had to overcome similar problems as the researchers above. The closest research to our process of testing security-configuration guides is the work of Spichkova et al. [25]. Their tool VM2 creates VM images and hardens them automatically with given security-configuration guides. They also use the CIS's guides, but they see guides as given and immutable, whereas we include in our approach the constant update and maintenance of the guides to adjust them to a company's security policy. Furthermore, they focus on the combination of Linux-based OSs and Ansible. In contrast, the diversity of technologies within Siemens forced us to support different application and check modes in our approach.

6 CONCLUSION

We have developed an approach towards authoring and maintaining machine-readable security-configuration guides that allows us to extend the DevOps principle of Continuous Integration to this domain. We achieved this by creating the Scapolite format that enables authors to combine human-readable information with machine-readable information on security-configuration requirements. The latter then serve as input for a process that (1) automatically generates artifacts for implementation and checking and (2) tests the created artifacts. Because the authors can specify the rules on an abstract level and thus do not have to manage such artifacts in parallel, we could significantly reduce the risk of errors because of manual errors and inconsistencies.

Due to the high degree of automation in our proposed process, we test the security-configuration guides and their generated artifacts much more frequently during authoring and maintenance than in the normal case. As a result, we detect the majority of problems before the release of a security-configuration guide.

In summary, our approach to security hardening via machine-readable security-configuration guides combined with the automated testing allows us to publish automated, well-tested mechanisms for implementing and checking along with the guide. Consequently, compliance with these configurations can be reached in a more timely and less error-prone manner, leading to better-secured systems.

REFERENCES

[1] V Atlidakis, P Godefroid, and M Polishchuk. 2020. Checking Security Properties of Cloud Service REST APIs. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. 387–397. https://doi.org/10.1109/ICST46399.2020.00046

[2] M Ceccato, D Corradini, L Gazzola, F M Kifetew, L Mariani, M Orrù, and P Tonella. 2020. A Framework for In-Vivo Testing of Mobile Applications. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. 286–296. https://doi.org/10.1109/ICST46399.2020.00037

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

[4] D Clerissi, G Denaro, M Mobilio, and L Mariani. 2020. Plug the Database Play With Automatic Testing: Improving System Testing by Exploiting Persistent Data. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 66–77.

[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] P Derakhshanfar, X Devroey, A Zaidman, A van Deursen, and A Panichella. 2020. Good Things Come In Threes: Improving Search-based Crash Reproduction With Helper Objectives. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 211–223.

[7] 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

[8] M C Gerten, J I Lathrop, M B Cohen, and T H Klinge. 2020. ChemTest: An Automated Software Testing Framework for an Emerging Paradigm. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 548–560.

[9] 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

[10] undefinedsmet Burak Kadron, Nicolás Rosner, and Tevfik Bultan. 2020. Feedback-Driven Side-Channel Analysis for Networked Applications. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2020)*. Association for Computing Machinery, New York, NY, USA, 260–271. https://doi.org/10.1145/3395363.3397365

[11] 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

[12] J W. Lin, N Salehnamadi, and S Malek. 2020. Test Automation in Open-Source Android Apps: A Large-Scale Empirical Study. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1078–1089.

[13] P X Mai, F Pastore, A Goknil, and L Briand. 2020. Metamorphic Security Testing for Web Systems. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. 186–197. https://doi.org/10.1109/ICST46399.2020.00028

[14] M Nass, E Alégroth, and R Feldt. 2020. On the Industrial Applicability of Augmented Testing: An Empirical Study. In *2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. 364–371. https://doi.org/10.1109/ICSTW50294.2020.00065

[15] opencontrol [n. d.]. OpenControl. http://opencontrol.cfapps.io/. Accessed: 2019-02-07.

[16] OVAL Board. [n. d.]. OVAL Documentation. https://ovalproject.github.io/. Accessed: 2021-04-16.

[17] A Perera, A Aleti, M Böhme, and B Turhan. 2020. Defect Prediction Guided Search-Based Software Testing. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 448–460.

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

[19] Markus Raab, Bernhard Denner, Stefan Hahnenberg, and Jürgen Cito. 2020. Unified Configuration Setting Access in Configuration Management Systems. In *Proceedings of the 28th International Conference on Program Comprehension (ICPC '20)*. Association for Computing Machinery, New York, NY, USA, 331–341. https://doi.org/10.1145/3387904.3389257

[20] 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

[21] Red Hat, Inc. [n. d.]. Ansible Documentation: working with Playbooks. https://docs.ansible.com/ansible/latest/user_guide/playbooks.html. Accessed: 2019-02-07.

[22] Red Hat, Inc. 2014. ComplianceAsCode. https://github.com/ComplianceAsCode/. Accessed: 2021-04-16.

[23] C Richter and H Wehrheim. 2020. Attend and Represent: A Novel View on Algorithm Selection for Software Verification. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1016–1028.

[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] Maria Spichkova, Biao Li, Lachlan Porter, Luke Mason, Ye Lyu, and Yi Weng. 2020. VM2: Automated security configuration and testing of virtual machine images. *Procedia Computer Science* 176 (2020), 3610–3617. https://doi.org/10.1016/j.procs.2020.09.025 Knowledge-Based and Intelligent Information & Engineering Systems: Proceedings of the 24th International Conference KES2020.

[26] Patrick Stöckle, Bernd Grobauer, and Alexander Pretschner. 2020. Automated Implementation of Windows-Related Security-Configuration Guides. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering* (Virtual Event, Australia) *(ASE '20)*. Association for Computing Machinery, New York, NY, USA, 598–610. https://doi.org/10.1145/3324884.3416540

[27] 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

[28] 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

[29] E Viglianisi, M Dallago, and M Ceccato. 2020. RESTTESTGEN: Automated Black-Box Testing of RESTful APIs. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. 142–152. https://doi.org/10.1109/ICST46399.2020.00024

[30] David Waltermire, Stephen Quinn, Harold Booth, Karen Scarfone, and Dragos Prisaca. 2018. The Technical Specification for the Security Content Automation Protocol (SCAP) Version 1.3. https://doi.org/10.6028/NIST.SP.800-126r3

[31] Y. Wang, M. Pyhäjärvi, and M. V. Mäntylä. 2020. Test Automation Process Improvement in a DevOps Team: Experience Report. In *2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. 314–321. https://doi.org/10.1109/ICSTW50294.2020.00057

[32] 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

[33] 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

[34] 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

[35] 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