Better Safe Than Sorry! Automated Identification of Functionality-Breaking Security-Configuration Rules

Patrick Stöckle*, Michael Sammereier*, Bernd Grobauer† and Alexander Pretschner*
*Chair of Software and Systems Engineering
Technical University of Munich (TUM)
Munich, Germany
Email: {patrick.stoeckle, michael.sammereier, alexander.pretschner}@tum.de
ORCID: 0000-0003-0193-5871, 0000-0002-5786-9554, 0000-0002-5573-1201
†T CST
Siemens AG
Munich, Germany
Email: bernd.grobauer@siemens.de, ORCID: 0000-0003-0792-3935

***Abstract*—Insecure default values in software settings can be exploited by attackers to compromise the system that runs the software. As a countermeasure, there exist security-configuration guides specifying in detail which values are secure. However, most administrators still refrain from hardening existing systems because the system functionality is feared to deteriorate if secure settings are applied. To foster the application of security-configuration guides, it is necessary to identify those rules that would restrict the functionality.**

This article presents our approach to use combinatorial testing to find problematic combinations of rules and machine learning techniques to identify the problematic rules within these combinations. The administrators can then apply only the unproblematic rules and, therefore, increase the system's security without the risk of disrupting its functionality. To demonstrate the usefulness of our approach, we applied it to real-world problems drawn from discussions with administrators at Siemens and found the problematic rules in these cases. We hope that this approach and its open-source implementation motivate more administrators to harden their systems and, thus, increase their systems' general security.

***Index Terms*—Software Security, Configuration Management, Software Testing**

I. Introduction

Researchers showed that in 15% of all data breaches, the attackers exploited cloud misconfigurations [1]. Nevertheless, there are various factors – mainly lack of knowledge – that prevent administrators from configuring their systems securely [2]. For most systems, we can resolve the *Lack of knowledge* by using a security-configuration guide from independent organizations like the Center for Internet Security (CIS). A guide is a set of rules and every rule specifies to which value the administrator has to set a configuration setting to make the system more secure. However, administrators are very reluctant to apply this configuration hardening to systems in production. Even if a guide is available and the necessary processes and tools to implement guides efficiently, they are discouraged by the fear of breaking the existing functionality. In this article, we focus on this problem of hardening existing systems and detect functionality-breaking rules automatically.

A. Motivating Example

Figure 1 shows the current situation of hardening existing systems. Administrator Alice is responsible for a server running essential business functions (see Figure 1, *Scenario*). In reality, these functions are automatic tests in different levels of abstraction from unit tests to end-to-end tests.

Alice wants to configure the server securely and uses a CIS guide. This guide has more than 500 rules. Alice automatically checks how many rules of the guide the system is currently not compliant with (see Figure 1, *Step 1*).

A recent study showed that a system using Windows 10 or Microsoft Office in the default configuration is, on average, only compliant with $\approx$ 17.7% of the corresponding CIS rules [3]. Thus, the checks will report more than 410 non-compliant rules. Alice could go through these rules and make for each rule two decisions (see Figure 1, *Step 2a*): First, is this rule important for the security of her server? A specific rule might be beneficial in general, but the threat addressed by the rule might not be relevant for her server. Second, could the rule interfere with the current behavior of the system, i.e., is the rule disabling some functionality the existing systems on the server still need?

Both decisions are complex and time-consuming. For the first one, we need a threat analysis to identify the relevant assets and resulting threats. For the second one, we need to know the potential side effects. For each side effect, we then have to check whether it affects the software running on the server. Although the rules include a description of the potential side effect, one must know how the existing software works to estimate potential problems with the rules to apply. If we calculated with one minute per decision, the process would last more than six hours. There are only a few systems where such an extensive analysis is economically reasonable. Thus, Alice has two choices. Either she applies all non-compliant rules on the server or does not harden the server.

If she chooses to apply all non-compliant rules, most probably problems in the system functionality will arise, and such

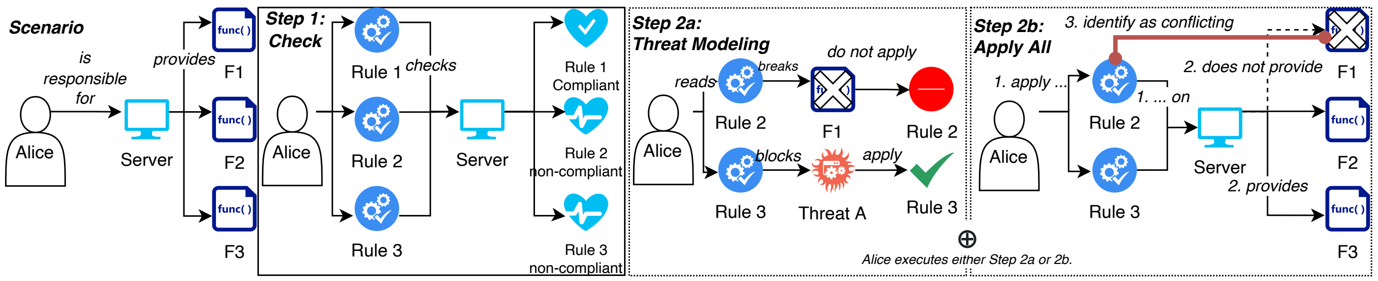

Fig. 1: The current process of hardening existing systems.

problems will be revealed by re-running regression tests after the hardening process. Thus, we assume for our example that there are broken functionalities. If Alice knows the software, she can guess which rules might cause the problems (see Figure 1, *Step 2b*); we call these rules functionality-breaking or just **breaking** rules. Otherwise, she has to disable rules until she finds all breaking rules.

If Alice found all the breaking rules and applied all other rules, she could guarantee the system's functionality and maximize the security gained by the configuration hardening. Usually, Alice does not know how many rules she has to exclude and how the rules work together. Multiple rules can break the same functionality, so she excludes all of them. Furthermore, she might not have to remove all breaking rules, as combining multiple rules breaks one functionality, so she excludes one of the corresponding rules.

When we applied a CIS guide to a test system at Siemens, we had to exclude 9 of the 500 rules, i.e., if we knew the number of rules to exclude, we would *only* have $\binom{500}{9}$ candidates. In practice, we do not need to investigate that many candidates, but, on average, this will cost far more time than the decision process, rendering this approach even more costly than the other.

Therefore, Alice will implement only rules for which she is 100% sure that they do not break a function or not harden the system at all. This behavior of dodging the risk of problems with the software functionality by neglecting the security configuration is widespread. We conducted a case study with two municipalities in Southern Germany: Although a guide existed for those municipalities, their systems only fulfilled 12% and 35% of the rules, respectively. They argued that they have a very heterogeneous environment and high availability requirements and, thus, did not want to tamper with the system's functionality. One might see this only as anecdotic evidence, but we heard the same argumentation from administrators at the Technical University of Munich.

B. Problem and Proposed Solution

The main problem addressed in this article is the following: We want to harden an existing system without interfering with its current functionality. Therefore, we want to find for a given guide, a given system and its given functionality a maximal subset of rules that does not break the functionality of the system.

To address this problem, we use existing combinatorial testing approaches combined with decision trees to efficiently find such a maximal subset (see Figure 2). First, we generate covering arrays based on the given rules. Second, we apply the rules specified in the current array entry, test all the functions, and store the corresponding result. Third, we train decision trees on the data from the previous step. Fourth, we use the shortest path leading to all functions working to find a set of breaking rules that leads us to maximal subset that does not break the functionality.

Our contribution is that we transfer established techniques from the software engineering, or more specifically, the software testing domain, to the security configuration domain to solve a widespread problem. Additionally, we share our code so practitioners can use it to find breaking rules.[1]

C. Definitions and Concepts

We use the motivating example to illustrate the definitions used in this article. We denote systems in general with ξ and Alice's server with ξ_A. We denote rules as $r \in \mathcal{R}$ and guides as set of rules, i.e., $\mathcal{G} = \{r_0, \ldots, r_n\}$. Furthermore, we abstract the functions as a set of predicates, i.e., $\mathcal{F} = \{f_1, \ldots, f_n\}$ in general and the functions of Alice's server in particular as $\mathcal{F}'$.

$$f(\xi, t) \Leftrightarrow \text{System } \xi \text{ fulfills the function } f \text{ at time } t$$

Here and in following formulas, we use the time as parameter to differentiate between different states of a system ξ. For example $f(\xi, t)$ and $\neg f(\xi, t+1)$ means that ξ fulfills f at t, but one step later, e.g., after apply some rules, not anymore. Thus, the server ξ_A in our example has to fulfill $\mathcal{F}'$. Next, we denote the check with $\aleph$.

$$\aleph(\xi, t, r) \Leftrightarrow \text{System } \xi \text{ is compliant to rule } r \text{ at time } t$$

Consecutively, we denote with the $\mathcal{R}^{NC}_{\mathcal{G},\xi,t}$ all non-compliant rules of $\mathcal{G}$ on system ξ at time t.

$$\mathcal{R}^{NC}_{\mathcal{G},\xi,t} = \{r | r \in \mathcal{G} : \neg\aleph(\xi, t, r)\}$$

In our example case, we would have $|\mathcal{R}^{NC}_{\mathcal{G}_{CIS},\xi_A,t_{Start}}| \geq 410$. Next, we denote the broken functionalities of $\mathcal{F}$ on the system ξ at time t with $\mathcal{F}^{F}_{\mathcal{F},\xi,t}$.

[1]GitHub.com/tum-i4/Better-Safe-Than-Sorry

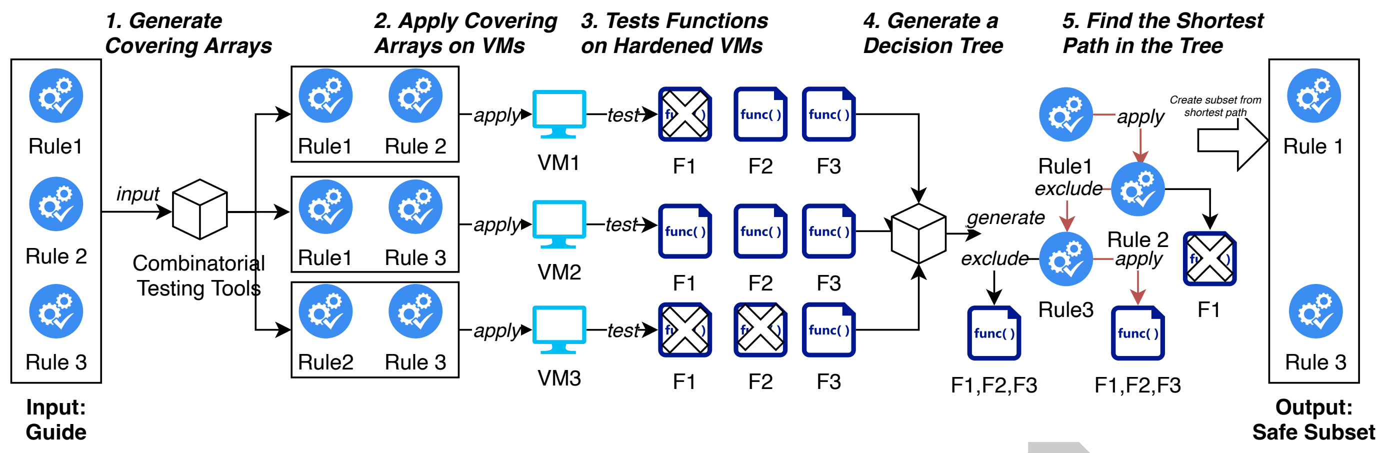
1. Generate Covering Arrays
2. Apply Covering Arrays on VMs
3. Tests Functions on Hardened VMs
4. Generate a Decision Tree
5. Find the Shortest Path in the Tree
Rule1
Rule 2
Rule 3
Input: Guide
input
Combinatorial Testing Tools
Rule1
Rule 2
Rule1
Rule 3
Rule2
Rule 3
apply
VM1
VM2
VM3
test
func( )
F1
F2
F3
generate
Rule1
exclude
apply
Rule 2
apply
exclude
Rule3
F1,F2,F3
F1,F2,F3
F1
Create subset from shortest path
Rule 1
Rule 3
Output: Safe Subset

Fig. 2: Identifying breaking rules using combinatorial testing and decision trees.

$$\mathcal{F}^{F}_{\mathcal{F},\xi,t} = \{f | f \in \mathcal{F} : \neg f(\xi,t)\}$$

Again, applied on our example, we would have $|\mathcal{F}'^{F}_{\mathcal{F}',\xi_A,t_{Start}}| \geq 1$ because there is a broken function. Additionally, we denote with b that a set of rules $\mathcal{R}$ is breaking, i.e., there is a failing functionality f after we applied all rules in $\mathcal{R}$.

$$b(\xi,t,\mathcal{R},\mathcal{F}) \Leftrightarrow (((\bigwedge_{f\in\mathcal{F}} f(\xi,t)) \wedge (\exists r \in \mathcal{R} : \neg\aleph(\xi,t,r)) \\ \wedge (\bigwedge_{r\in\mathcal{R}} \aleph(\xi,t+1,r))) \Rightarrow \exists f \in \mathcal{F} : \neg f(\xi,t+1))$$

We denote the breaking rules of $\mathcal{G}$ on a system ξ at a time t with respect to the functionalities $\mathcal{F}$ with $\mathcal{B}_{\mathcal{G},\xi,t,\mathcal{F}}$.

$$\mathcal{B}_{\mathcal{G},\xi,t,\mathcal{F}} = \{r | r \in \mathcal{G} : b(\xi,t,r,\mathcal{F})\}$$

Similarly, we define with n for a set of rules $\mathcal{R}$ that they are **non-breaking** on a system ξ at a time t with respect to the functions $\mathcal{F}$.

$$n(\xi,t,\mathcal{R},\mathcal{F}) \Leftrightarrow \\ \left(\bigwedge_{f\in\mathcal{F}} f(\xi,t) \wedge \bigwedge_{r\in\mathcal{R}} \aleph(\xi,t+1,r)\right) \Rightarrow \bigwedge_{f\in\mathcal{F}} f(\xi,t+1)$$

Based on the definition of n, we can now define a maximal non-breaking set: A subset $\mathcal{G}^*$ is a maximal non-breaking set with respect to the system ξ, at the time t, and the functions $\mathcal{F}$ if $\mathcal{G}^*$ is non-breaking and any additional rule r' added to $\mathcal{G}^*$ would break at least one functionality.

$$\mathcal{G}^* \text{ maximal non-breaking with } \mathcal{G},\mathcal{F},t \Leftrightarrow \mathcal{G}^* \subset \mathcal{G} \\ \wedge n(\xi,t,\mathcal{G}^*,\mathcal{F}) \wedge \forall r' \in \mathcal{G}\setminus\mathcal{G}^* : \neg n(\xi,t,\mathcal{G}^* \cup \{r'\},\mathcal{F})$$

Therefore, the goal of our approach is to find for a given system ξ, given functions $\mathcal{F}$, and a given guide $\mathcal{G}$ at the time t a maximal non-breaking set $\mathcal{G}^*$.

II. Generate Covering Arrays From Security-Configuration Guides

The naive approach to solving our problem would be to test every possible combination of rules $\mathcal{P}(\mathcal{G})$. We could search the combinations without failing tests for the set with the most applied rules, but this is a very inefficient approach. In the domain of software testing, researchers have already solved a similar problem: If we want to test a program with many different parameters, we want to test it in all possible combinations of the parameters. We can use combinatorial testing to test programs *enough* without testing all the parameter combinations [4]. Depending on the desired strength, we can drastically reduce the combinations to test with combinatorial testing compared with testing all combinations. However, we can only reliably detect all faults up to this level, i.e., combinatorial testing of strength 2 can detect all faults caused by the combination of two or fewer parameters [5].

If we apply the combinatorial testing approach, we need to decide for a targeted strength for finding breaking rules in a guide. Since there is no data from the security-configuration domain, we have to rely on data from software testing. A study by Kuhn et al. found no failing tests with a combination of more than six parameters [4]. Thus, we *assume* that there is **no** combination of **more than six rules** causing a function to break. However, we discuss this strong assumption later in section VIII.

To apply combinatorial testing in the security-configuration domain, we generate once for every guide $\mathcal{G}$ with $n = |\mathcal{G}|$ rules a set of n-tuples with `true` or `false`; `true` at the position i of a tuple means we apply the rule i in this combination. In combinatorial testing, such a tuple is called Covering Array.

```
[System]
Name: all_rules
[Parameter]
R1_1_1 (boolean) : true, false
R1_1_2 (boolean) : true, false
```

Listing 1: CIS guide transformed into the ACTS input format.

```
# Degree of interaction coverage: 2
# Number of parameters: 507
# Number of configurations: 20
R1_1_1,R1_1_2,R1_1_3,R1_1_4,R1_1_5,R1_1_6,...
true,true,true,true,false,false,...
true,true,false,false,true,true,...
false,false,true,true,true,true,...
false,false,false,false,false,false,...
...
```

Listing 2: ACTS output for a CIS guide.

We can reuse these covering arrays for multiple systems with different functionalities to find breaking rules. If we add or remove new rules, we must regenerate the covering arrays. Since the guides contain hundreds of rules, we needed algorithms that could handle tuples of that size. Thus, we use the IPOG [6] and IPOG-D [7] algorithms and their implementations in the Automated Combinatorial Testing for Software (ACTS) tool to generate the combinations [8].

We first translate the rules of a guide from their original format into an ACTS input file defining the used parameters. In this input file, each parameter has a name and a data type, i.e., in our scenario, the rule's ID, e.g., `R1_1_1`, and `boolean`. One can see an example in Listing 1. Depending on the chosen degree of covering arrays and algorithm, ACTS now generates the covering arrays. One can see an example output in Listing 2. Afterward, we translate the ACTS output into JSON files we use to implement guides automatically [3]. We included several versions of these JSON files for IPOG and IPOG-D and different degrees in our repository. Furthermore, one can use our published Python code to replicate these steps on their own. After this first step, we now have the different covering arrays of the rules in our guide in a form we can automatically implement on a system to test tuple break the system's functionalities.

III. Tests Functions on Hardened Instances

Depending on the degree of the covering arrays, we generate between 20 and 5545 tuples for the CIS Windows 10 guide. In the next step, we apply every tuple of rules and use the given tests $\mathcal{F}$ to check if all functions are still accessible. If a test fails, we record this in a log file. After we have executed this procedure for every tuple, we collect the different log files and merge them. The resulting file states which tuple broke a test and which did not.

What sounds straightforward, in theory, was cumbersome to realize. The first problem was that we had to prepare an environment where we could set up the software, apply all rules in a tuple, test the functionalities, and record the result. To solve this problem, security experts at Siemens use a toolchain with Ansible, Vagrant, and AWS instances to efficiently provision several virtual machine (VM) instances that they can configure independently and in parallel [9]. An alternative is to use Vagrant and a hypervisor like VirtualBox to run the VMs locally. The second problem was ordering effects, i.e., a test is failing not because of the currently applied tuple but the previous one. To avoid these effects, we could reset the VM after every test run or do a soft reset where we only revert the applied rules. The hard reset costs more time than the soft reset, but there is no risk of side effects, e.g., if the revert mechanism of a rule does not work. With the code in our repository, one can generate one VM instance for every tuple or several instances and distribute the tuples uniformly over the instances. The third problem was the automatic tests. Ideally, we would use tests from the industry to test real-world functionality. However, as we stated before, only a few companies apply configuration hardening to their systems. Those companies manually check whether their systems still work after the configuration hardening. Thus, we needed to create our automatic tests ourselves. Nevertheless, if all tests pass although the rules break the functionality, we will not detect this in our testing process, but only on the productive systems; it is, therefore, crucial to find suitable tests [10].

```
[{ "name": "custom_1", "breaking": false,
   "rules": ["R1_1_1", "R1_1_2", "R1_1_3", "..."]},
{  "name": "custom_2", "breaking": true,
   "rules": ["R1_1_1", "R1_1_2", "R1_1_5", "..."]},
"..."]
```

Listing 3: Example results of the testing process.

```
instances/vagrant_0
├─ Vagrantfile                # Includes the used Vagrant box
├─ cis_windows_10_1909        # The guide to apply
│   ├── sfera_automation.json # Export JSON with all the
│   │                         # rules and CA tuples.
│   └── sfera_automation.ps1  # Library to apply the guide
├─ setup.ps1                  # Script for the test process
└─ tests                      # Test directory
    ├── test.ps1              # Script coordinating the tests
    ├── tst_acc_cr.ps1        # Example test
    └── tst_conf.json         # Test configuration, e.g.,
                              # which tuples should we
                              ↪  execute
```

Listing 4: Example Vagrant directory.

After solving those issues, our testing procedure consists of the following steps:

1) Prepare the image of ξ: First, we prepare an image, i.e., a Vagrant box, with our software and all needed dependencies as reference to set up the different instances.
2) Prepare the instances ξ_i: We prepare for every VM instance ξ_i to start a directory with the software to run, the tests, the guide, and the CA tuples to test. If we want to test several tuples on one instance, we distribute the tuples uniformly. One can see the layout of such a directory in Listing 4. On the one side, we can fasten the instance setup if we add more actions into the box setup. On the other side, we want the image to be flexible and fit more than one system, thus keeping it slim.
3) Start the instances ξ_i: We start the instances either in parallel or sequentially.
4) Is the system working?: We execute all tests $\mathcal{F}$ in one instance ξ_0 to see whether the functionality works on an instance in the default configuration. If the tests fail before applying any rules, i.e., $|\mathcal{F}^F_{\mathcal{F},\xi_0,t_0}| \geq 1$, there must be a problem in the tests or the setup that we need to fix.

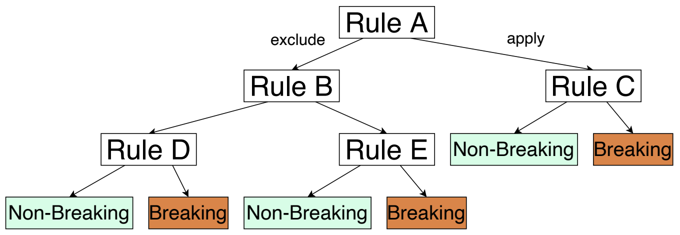

Fig. 3: Simplified example of a generated decision tree.

5) Apply all rules in $\mathcal{G}$: We apply all rules on one instance ξ_0, i.e., $\mathcal{R}^{NC}_{\mathcal{G},\xi_0,t_1} = \emptyset$
6) Run all tests in $\mathcal{F}$: If there is a breaking rule in the guide, we will see at least one test failing, i.e., $|\mathcal{F}^F_{\mathcal{F},\xi_0,t_1}| \geq 1$. If no tests fail, i.e., $\mathcal{F}^F_{\mathcal{F},\xi_0,t_1} = \emptyset$ we can safely apply all rules and stop the testing process at this point.
7) Revert all rules: If we use the soft reset, the revert mechanism must work, i.e., $\mathcal{R}^{NC}_{\mathcal{G},\xi_0,t_2} = \mathcal{R}^{NC}_{\mathcal{G},\xi_0,t_0}$. However, there are some rules that we cannot revert. Thus, we try to revert all rules and execute the tests again. If some tests still fail, i.e., $|\mathcal{F}^F_{\mathcal{F},\xi_0,t_2}| \geq 1$, there is a problem with the revert mechanism.
8) Apply a tuple: We take the first tuple of the list of untested tuples and apply all rules $\mathcal{G}_j \subset \mathcal{G}$ corresponding to this tuple on an instance ξ_i, i.e., $\mathcal{R}^{NC}_{\mathcal{G},\xi_i,t'_j} = \mathcal{R}^{NC}_{\mathcal{G},\xi_i,t_0} \setminus \mathcal{G}_j$ with t' being after the application.
9) Test the functionality $\mathcal{F}$: We execute the tests and store whether there were failing tests, i.e., $|\mathcal{F}^F_{\mathcal{F},\xi_i,t'_j}| \geq 1$, in a JSON file.
10) Reset: To prepare a clean environment again, we perform a reset so that $\mathcal{R}^{NC}_{\mathcal{G},\xi_i,t''} = \mathcal{R}^{NC}_{\mathcal{G},\xi_i,t_0}$. Afterward, we go back to Step 7 until we have applied all tuples.
11) Collect the results: After we have applied and tested all tuples, we collect all results, i.e., $\mathcal{G}_j$ and whether there were failing tests $|\mathcal{F}^F_{\mathcal{F},\xi_i,t'_j}| \geq 1$, from the different instances ξ_i and combine them in a single JSON file. One can see an example output in Listing 3.
12) Tear down: We destroy the instances ξ_i.

Again, one can use our published code to redo these steps on their own. We have now tested all tuples, and the resulting JSON states which tuples cause which tests to fail. In the next step, we use this information to deduce which rules caused the failures.

IV. Analyze the Test Results

As a result of the previous step, we know for each of the tuples whether the application of these rules broke the functionality or not (see Listing 3). Next, we analyze these data to find a maximal non-breaking set $\mathcal{G}^* \subset \mathcal{G}$. We could pass the failing tuples to the administrators like Alice so that they adjust the software on ξ to work even when these tuples are applied. However, especially for legacy systems, it may be challenging to carry out changes; Therefore, we want to adjust the guide by removing potential problematic rules for our general scenario. At Siemens, the administrators would decide whether they need other security measures to reduce the risks resulting from the excluded rules.

We can visualize the results of the previous step in a truth table. Every row is a CA tuple $\mathcal{G}_j$, every column is a rule $r \in \mathcal{G}$, and the last column is the result of the tests, i.e., $\mathcal{F}^F_{\mathcal{F},\xi_i,t'_j} \geq 1$. Finding a minimal cutting set in such a structure is a common task in different computer science disciplines, and there are different solutions. We used two different approaches for our proof of concept.

A. Decision Trees

First, we adapted the approach described by Yilmaz et al. [11] for our scenario. They used machine learning to find the parameters causing a test to fail and trained a decision tree on their test results. Thus, we ported their approach into our domain and learned decision trees on our results.

One can see an example decision tree in Figure 3. The nodes in the tree represent rules from the tuples. The algorithm calculated different partitions of breaking and non-breaking tuples by checking whether we apply a rule r in $\mathcal{G}_j$, i.e., $r \in \mathcal{G}_j$, or not. In Figure 3, the algorithm differentiates first between tuples where *Rule A* is applied or not. Every leaf node states whether the functionality was broken, i.e., $|\mathcal{F}^F_{\mathcal{F},\xi_i,t'_j}| \geq 1$, or not when we applied or excluded the rules on the path between the root and the leaf node. In Figure 3, applying *Rule A* but excluding *Rule C* leads to a non-breaking leaf, i.e., *Rule C* is a breaking rule.

We could use this decision tree to predict whether a tuple that we have not tested before is breaking or not. However, we are only interested in the non-breaking leaves with the least excluded rules. Therefore, we use shortest path search algorithms like Dijkstra's algorithm on the decision trees to find these leaves. We give edges that apply a rule, i.e., right edges in Figure 3, a value 0, and all other edges the value 1. One can see an example in Figure 4. Therefore, the total cost of a path from the root to a leaf is the number of rules that we did not apply. In turn, the shortest path leading to a non-breaking leaf is the path with the least number of rules not being applied, i.e., a maximal non-breaking set $\mathcal{G}^*$. In Figure 4, this is $\mathcal{G}^* = \mathcal{G} \setminus \{R_C\}$.

We implemented our approach using *scikit-learn* [12]. First, we translate the data from the previous step into the scikit format. Second, we train a decision tree on it. Third, we added the weights to the learned decision tree. Fourth, we run a shortest path algorithm on the weighted decision tree to find a non-breaking leaf. Five, we follow the path from the root to this leaf. If the current edge has the weight 1, we remove the current rule from the guide $\mathcal{G}$. The resulting set is a maximal non-breaking set $\mathcal{G}^*$.

B. Logic Minimization

In addition to the heuristic approach, we can formally examine the results with logic minimization to derive the minimal set. Here, we pass our truth table representing our test results to one of the many minimization algorithms [13]

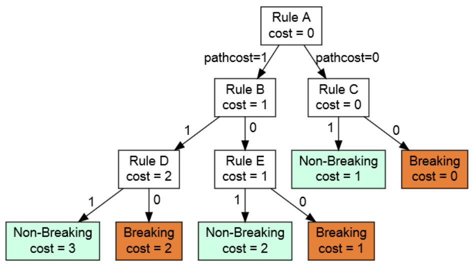

Fig. 4: Visualization of the modified decision tree from Figure 3 with weights to support shortest path search.

and use the minimized table to find the maximal non-breaking set $\mathcal{G}^*$.

We first transform the test results into a truth table in our implementation. Second, we pass this table to the minimization library *PyEDA* [14]. Third, we exclude the rules according to the minimized table to derive $\mathcal{G}^*$.

We provided sample results and code in our repository so that one can redo these steps with both approaches. In the end, we have a maximal non-breaking set $\mathcal{G}^* \subset \mathcal{G}$, and we can use it to harden our system safely without breaking any legacy functionality.

V. Evaluation

When evaluating our approach and its implementation, we focus on the following research questions:

RQ1 Can we find a maximal non-breaking subset $\mathcal{G}^*$ of a guide $\mathcal{G}$ with respect to given functionalities $\mathcal{F}$ using combinatorial testing? What degrees of breaking rules can we detect? In theory, the strength of the covering array should be the upper limit for the degree of breaking rules we can detect. However, the combination with the decision trees could reduce our approach's power in practice.

RQ2 How much time does our approach need? Is this a reasonable time effort for hardening a system?

We used the CIS guide for Windows 10 version 1909 [15] for our evaluation. This guide contains 507 rules, i.e., $|\mathcal{G}_{CIS,W10}| = 507$. To apply and revert the rules automatically, we transformed the guide into the Scapolite format [3]. In the following, we discuss how we evaluated the different steps of our approach.

A. Evaluation of the Covering Array Generation

First, we had to evaluate the generation of the covering arrays from an existing guide. As discussed in section II, we wanted to generate covering arrays with strength from 2 to 6 with both IPOG and IPOG-D. We evaluated how long the generation takes for the $\mathcal{G}_{CIS,W10}$ guide to partially answer **RQ1**.

```
[["R1", "R2"], ["R3", "R4"]]
```

Listing 5: Example definition of a breaking combination.

TABLE I: Number of generated tuples depending on the used algorithm and strength.

Algorithm \ Strength	2	3	4	5
IPOG	20	70	209	-
IPOG-D	20	78	305	5545

We run the ACTS tool on a server with two Intel Xeon E5-2687W v3 CPUs with 40 cores and a total of 500 GB of RAM, from which we used up to 340 GB.

B. Testing Process

We evaluated our testing process in three steps.

1) Simulation: In the first step, we simulated the breaking rules. Here, we first defined combinations of rules that break the functionality. We defined 51 combinations of breaking rules; one combination is the empty set, i.e., the special case where we can apply our guide without problems. We define them as logical formulas in disjunctive normal form; Listing 5 shows how we express $(R1 \wedge R2) \vee (R3 \wedge R4)$. If we apply all rules of the first or the second subformula, the system's functionality will break, e.g., we can apply `R1` and `R2` to break the function, but not `R1` and `R3`. Furthermore, we tested for each of the non-empty combinations 3 additional random variants, i.e., we replaced the rules' ids with random other ids to avoid potential side effects based on the order of the rules. In total, we thus tested 201 sets; one can find all 201 sets in our repository to reproduce our evaluation.

For each of the combinations, we then went through the list of tuples and created for each tuple the result file: If a CA tuple includes a breaking combination, we mark the tuple as breaking else as non-breaking. Afterward, we combine all result files and analyze them using our decision-tree-based and logic minimization approach. We used different covering arrays from both generation algorithms and with different strengths.

2) Validation of the Simulation: In the second step, we evaluated the complete process but used generated mock tests based on the defined combinations from the Evaluation Step 1. As described in section III, we execute all steps. However, the tests in Step 8 check for a given definition of breaking rules and whether the current system ξ is compliant with those rules. If so, we mark the current tuple as failing. We could test many combinations of breaking rules with those generated mock tests. In contrast to the simulation, this step required significantly more time. While executing the tests is cheap, setting up the VM and then applying and reverting the rules takes some time. We conducted this part of the evaluation with the covering arrays of strength 4 generated by the IPOG-D on two VMs.

3) Practical Application: In the third step, we evaluated the process with a real test. Here, we needed a small program f' that fails when the whole guide is applied, i.e., $\mathcal{R}^{NC}_{\mathcal{G}_{CIS,W10},\xi,t'} = \emptyset$. We chose a simple PowerShell script that creates a new

TABLE II: Time (in seconds) needed to generate covering arrays of given strength depending on the used algorithm.

Algorithm \ Strength	2	3	4	5
IPOG	0.7	374	179149	-
IPOG-D	0.2	16	8451	1184478

user with a password and then deletes the user again. This test symbolizes the nightmare of all administrators like Alice: It is a straightforward task that usually takes seconds but does not work after the hardening. Thus, they must spend hours investigating which rule broke the functionality that does not relate to the script's original time effort. Next, we did all the steps of our process with two instances running in parallel and marked the tuples as failing when f' failed. Again, we gave the result to our analyzing components to find $\mathcal{G}^*_{CIS,W10}$. In the end, we applied $\mathcal{G}^*_{CIS,W10}$ and checked whether f' was working or not. We again conducted this part of the evaluation with the covering arrays of strength 4 generated by the IPOG-D on two VMs.

The simulation helps us to answer **RQ1**. In Step 2 and Step 3, we measured the time needed since this contributes significantly to the answer of **RQ2**.

C. *Evaluation of the Analysis*

We need the analysis of the test results to assess the general quality of our approach. However, we also compared different factors: First, we compared the analysis variant, i.e., the decision-tree-based heuristic and the logic minimization approach. Second, we compared the influence of different parameters, e.g., for the decision tree generation. Third, we compared different algorithms to find a solution in the tree-based approach, e.g., taking the non-breaking leaf with the most samples.

We assessed for the different variants whether they calculated a correct maximal non-breaking set $\mathcal{G}^*$ and checked how much they differ in the case of incorrect output. Thus, we can answer **RQ1**. Moreover, we measured how long the calculation of the breaking rules takes depending on the input, contributing to the answer of **RQ2**.

VI. Results

In this chapter, we will present the results of the evaluation of our proof of concept.

A. *Results of the Covering Array Generation*

Table I shows the number of tuples per covering array for different strengths and both IPOG and IPOG-D. The number of tuples grows exponentially with increasing strength. However, due to the required time, we could not generate covering arrays of the strength of 5 using IPOG or 6 with IPOG-D. Table II shows the required time to generate the covering arrays. For all strengths between 2 and 4, the time is an average of 5 measurements; for 5, we measured the time only once. One can see that IPOG-D – as expected – is faster than IPOG. Thus, for IPOG-D, we could even calculate the covering arrays with strength 5, although it took ≈ 13.7 days. However, IPOG-D generated more tuples for covering arrays of the same strength.

B. *Results of the Testing Process*

1) Simulation: Figure 5 shows our simulation results with covering arrays of strength 2 to 5. We cluster the breaking rule set based on the number of clauses they have on the x-axis and what is the maximal number of rules within a clause on the y-axis. A red triangle means that our approach could not identify a single solution correctly for all sets in this cluster. An orange square means that our approach identified some solutions correctly for the sets in this cluster. A green diamond means that our approach identified all solutions correctly for this cluster. Based on the strength of the covering arrays, we expected a green triangle in the lower left corner. However, our approach performed worse than expected for 2-way and 3-way arrays but better for 5-way arrays. The most prominent reason is the lack of data for the decision tree. The decision tree has no data to partition and, therefore, cannot determine any rules to be excluded. In some cases, the algorithm could find subsets of the correct solutions. We consider these cases invalid results for this evaluation, but they may contribute to finding the optimal solution. Surprisingly, our approach could calculate the correct results for single clauses with up to 11 rules per clause based on the covering arrays of strength 5, although we only expected correct results for up to 5. We investigated the breaking rule sets that were not calculated correctly in more detail. Our approach could not determine the correct solution for combination in Listing 6 and all its variants. Instead of excluding one rule from each of the three clauses, our algorithm only excludes `R2_2_4` and `R2_2_7`, i.e., a subset of one correct solution. Thus, we investigated the corresponding decision tree. Although correct solutions were part of the tree, the wrong solutions dominated them because of the shorter length. We are not sure why the decision tree was partitioned, but maybe more test tuples would have helped. In general, our approach calculated the correct solution 77% of the breaking rules in our sample set. One could argue that this is too low to use the approach in practice, but our sample set includes far more complicated combinations than we expect in reality. On the realistic samples, based on the assumption that only up to 6 rules combined lead to a problem, we achieve almost 100%.

2) Validation of the Simulation: When we started this part of the evaluation, the initial tests failed, i.e., Step 4 of the testing process. These initial failing tests never happened for the simulation. The reason is those mentioned above $\approx 17.7\%$ compliant rules on a system in its default configuration. If all rules in a clause are already compliant with a default system, the mock test fails at Step 4. Thus, we had to skip those tests;

```
[["R2_2_1", "R2_2_2", "R2_2_3"],
 ["R2_2_4", "R2_2_5", "R2_2_6"],
 ["R2_2_7", "R2_2_8", "R2_2_9"]]
```

Listing 6: Problematic breaking rule set.

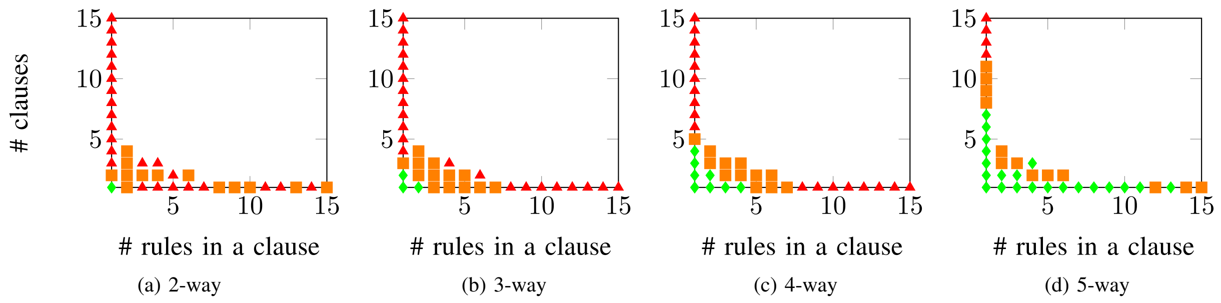

Fig. 5: Distribution of the breaking rules sets.

an alternative was to use an image non-compliant with all rules, but we deemed this not a realistic scenario. Apart from this, the generated mock tests lead to the same data as the simulation. The whole process took around 12 hours.

3) Practical Application: We evaluated the practical application of our proof of concept by testing functionality with unknown breaking rules. We knew from the initial tests that the functionality worked before applying any rules but did not work after applying them. Overall, the testing process took 12 hours to complete, resulting in 156 breaking and 149 non-breaking combinations. Figure 7 shows the decision trees learned on these results stating that one should exclude rule `R1_1_4`. Afterward, we applied $\mathcal{G}^* = \mathcal{G}_{CIS,W10} \setminus \{R1\_1\_4\}$, ran our test again, and it succeeded. Rule `R1_1_4` "ensure [that] 'Minimum password length' is set to '14 or more character(s)' ", but our test script tried to set a password of length 6 and thus failed.

C. Evaluation of the Analysis

Next, we compared different techniques to find the optimal solution. First, we select the non-breaking leaf with the biggest partition instead of the leaf with the shortest path. We developed this potential improvement based on the problems seen, e.g., in combinations like Listing 6. We used the 5-way combinations generated by IPOG-D to compare the different path algorithms. Figure 6 shows the result of the modified approach. One can see that the figure is almost identical to Figure 5, but the modified approach performed better in the mixed clusters like Listing 6, i.e., some orange triangles are here green diamonds.

Apart from modifying the used path algorithm, we also evaluated modifications of the generation of the decision tree, e.g., introducing a minimum for the number of test cases in a partition or changing the minimum number of data points for a split of the partition. However, none of these changes lead to identifying more correct solutions.

We also evaluated the logic optimization approach compared to the decision tree-based heuristic approach. The logical optimization took more time than the heuristic. Thus, we excluded breaking rules sets with more than 5 clauses, more than 9 rules per clause, and did not evaluate the random variants of breaking rules sets. Since more test cases increased the size of the truth table to be minimized, calculation time heavily depended on the strength of the covering arrays; the algorithm calculated the results of strength 2 and 3 in seconds, 4 in minutes, but for 5, it took more than 24 hours. Also, more clauses lead to longer computation. As expected, the logical optimization calculated all solutions correctly within the strength of the covering arrays and most of the remaining solutions; it incorrectly calculated only 3 of the 35. The decision tree-based heuristic led to 4 incorrect results in this sample set. Thus, the performance of the logic minimization approach is similar to the decision-tree-based heuristic.

VII. Discussion

In this chapter, we will discuss the results of the evaluation and answer our research questions.

A. Finding Breaking Rules

The results of our evaluation showed that one can use combinatorial testing and specifically covering arrays to find breaking combinations of rules. Furthermore, we can use logical minimization or machine-learning-based heuristics to find a maximal non-breaking set $\mathcal{G}^*$. However, our results also showed that there are some caveats. First, we could not create covering arrays with a strength of 6, although we assumed this was the necessary strength to cover all breaking sets in practice. Even generating covering arrays of a strength of 5 occupied too much memory (340GB) for (13 days) so this is hardly useful for public guides, e.g., from the CIS. For private guides, e.g., at Siemens, it is not economical to spend that many resources on every guide. Thus, it is more realistic to use 4-way covering arrays, guaranteeing that we will find all combinations of 4 breaking rules or less. Nevertheless, the results show that our heuristic approach can find some solutions for higher combinations or at least subsets of an optimal solution that could help the administrators. Since we have no information about the distribution of the breaking rules in practice, the answer of **RQ1** is twofold. If all or most breaking functionality results in practice from 4 or fewer rules, covering arrays and our heuristic analysis can reliably identify the breaking rules. If a significant portion of breaking functionality results from 5 or more rules, covering arrays and heuristic analysis can only help to identify the optimal solution.

B. *Effort*

Generating different covering arrays depends on the number of rules in the chosen guide and the desired strength of the combinations. Based on that, it requires a couple of hours, days, or even weeks. However, we need this generation only once when the publisher publishes the guide and not for every system we want to harden with the given guide. The CIS also updates its guides, but if the rules change, they do not have to regenerate the covering arrays. If they add new rules, they could reuse the existing arrays to speed up the generation. As stated above, we would argue that it is more realistic to use 4-way covering arrays as 5-way covering arrays are too expensive.

We can estimate the effort of identifying breaking rules for a given system ξ using the following formula:

$$t_{\Sigma} = N_{VMs} \cdot t_{VM} + t_{SW} + \frac{N_C}{N_{VMs}} (t_A + t_T + t_{SR}) + t_{ANA}$$

with

N_C	the number of tuples, e.g., 305 for the $\mathcal{G}_{CIS,W10}$ guide and 4-way covering arrays of the IPOG-D.
N_{VMs}	the number of instances, e.g., 2 in our evaluation.
t_{VM}	the time needed to prepare the instances, i.e., Testing process, Step 2.
t_{SW}	the time to start the instances and set up the software whose functionality we want to ensure, i.e., Step 3.
t_A	the time needed to apply all rules in a tuple, i.e., Step 8.
t_T	the time needed to execute the automatic tests, i.e., Step 9
t_{SR}	the time needed to do a soft reset of the applied rules, i.e., Step 10
t_{ANA}	the time needed for the analysis of the test runs

Setting up a local VM, e.g., with Vagrant, can last for several minutes, whereas a VM in the cloud, e.g., on AWS, is much faster. t_{SW} depends on the complexity of the software. In our evaluation, we tested a core function of Windows and, thus, did not install any additional software. In our previous study [3], we showed that t_A is for Windows rules below one 1s per rule but not negligible when executed many times. t_T depends on the complexity of the tests. The script took a couple of seconds in our evaluation, but if one uses complex tests, this could take minutes or hours. t_{SR} in contrast, is again in the order t_A. t_{ANA} with the decision trees, and the shortest path algorithm takes only a couple of seconds using our heuristic and several minutes with the logic minimization.

As mentioned in subsubsection VI-B3, our experiments run for more than 12h with covering arrays of strength of 4, answering partially **RQ2**. This time might be reasonable if we execute it only for releases, but it is too much to use in continuous integration contexts. We can mainly influence two factors in the equation: the number of tuples N_C and the number of instances N_I. If we reduce the number of tuples by choosing covering arrays with lower power, we will not detect some complex combinations. Thus, we will increase

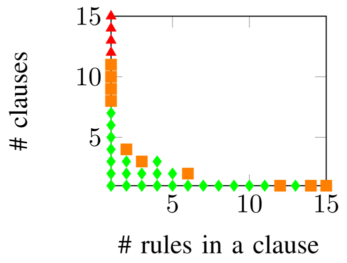

Fig. 6: Distribution of the breaking rules sets when using combinations of strength 5 and the modified approach.

N_I by using more instances in parallel, e.g., on-demand in the cloud. If we used 30 on-demand instances, we only ran 11 combinations on every machine. On average, a combination applied 240 rules, i.e., the application and software reset would last at most 240s. Assuming that the automatic tests last 2min, the whole process would need 2h. To estimate this process's price, we calculated 30 on-demand Windows instances, each with 2 cores, 4GB RAM, and 25GB storage on AWS. The estimation there was that this would cost around \$3. We argue that 2h is short enough to include this process into regression tests running every night and \$3 is cheap enough to execute the process for most systems.

VIII. Threats to Validity

In this section, we list potential threats to the validity of our results.

A. *Internal validity*

1) Choice of the Breaking Rule Sets: We evaluated our approach with our breaking rule sets. We tried to cover a variety of possible breaking rules and added, e.g., formulas with no overlap, such that the result must contain one rule from each subformula. However, there could still be an unintended bias in these sets.

B. *External validity*

1) Lack of Good and Automated Tests: One core assumption of this article is that there are automatic tests whose failure indicates that some function is broken. However, in most organizations, there are no tests for their legacy systems at all. Some organizations test their legacy systems manually, but even if the needed effort is minimal, repeating a manual task in combination with the covering arrays makes the whole process impractical. If there are automatic tests, they might not reveal that the functionality is broken. We would then harden the running system, break some functionality, but recognize this much later, probably with some outtakes and user complaints. Writing good tests is a hard problem, and we did not include the effort of creating such test cases in the effort calculations of our approach.

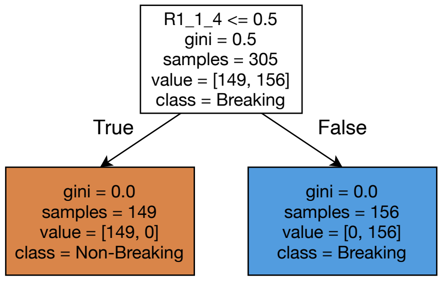

Fig. 7: Generated decision tree for the example functionality.

2) Lack of Real Breaking Rule Sets: Our chosen breaking rule sets might not represent breaking rules in real-world systems. We based our breaking rules sets on results on the literature claim that there are no failures resulting from combining more than 6 parameters. However, we do not know what combination of rules causes problems in practice or how their distribution is. Our assumption here is that the occurring problems might also depend on the size and complexity of the system, i.e., that more complex combinations of breaking rules only occur in more complex systems. We needed an extensive case study where we collected real-world configuration problems, identified the breaking rules manually, and assessed the complexity. With this case study's data, we could determine the distribution of the breaking rules and assess how useful our approach is in practice.

3) Problematic Proof of Concept Setup: We assume that one can use our approach with any guide. However, we show this with our proof of concept implementation only for Windows guides. Furthermore, we assumed that one can *isolate* the system in an image which might also not be the case for many systems, especially complex systems communicating with each other and the internet.

4) Importance of the Maximal Set: We claim that the algorithm is successful if it returns a maximal set of non-breaking rules for a given profile based on the current breaking rule set. However, this might not be that important in practice. If Alice applies all but one rule of the theoretical maximal set, the system's security might not be optimal, but much better than without hardening. Thus, a more efficient heuristic algorithm returning only a subset of a maximal set might be more economical in practice. Furthermore, not every rule is equally important as the other. Thus, we need to incorporate the importance of the rules into our approach. We need to consider in the future what the optimal solution is and what acceptable solutions are. The next step will be to adjust the weight of the rules based on the system and its criticality. The result here could also be that we only need a couple of essential rules on a specific system to reach a good level of security and that the gain of applying the complete guide might not be that high.

IX. Related Work

Research on configuration management is well-established, but there are constantly new insights into this topic [16], [17], [18], [19]. Dietrich et al. presented the best overview of the needs of administrators to avoid security misconfigurations [2]. The work presented in this article depends on previous articles in the context of configuration hardening addressing other essential factors such as the *Poor vendor documentation* [20]. Our approach needs automatically implementable guides as presented by [3]. Furthermore, we use techniques presented in [9] for the setup and testing of the VMs.

Although originating in the 80s [21], the combinatorial testing research is still very active [22]. Kuhn et al. showed that one could use combinatorial testing to test whether the software works with all settings of the software itself, but also whether the software works in every possible configuration of a system [23]. As mentioned before, we use the IPOG [6] and IPOG-D [7] algorithm to generate our covering arrays. The most inspiring article for our work was the approach Yilmaz et al. of to finding faults when testing software using combinatorial test cases [11]. Furthermore, they suggested further analyzing the test case results using classification tree analysis. We ported their approach to the domain of security configuration. However, they were only interested in a combination failing, whereas we are interested in a maximal solution, i.e., applying as many rules as possible.

X. Conclusion

We have shown in this article that one can use combinatorial testing to find combinations of breaking rules and machine-learning-based heuristics to find maximal non-breaking sets $\mathcal{G}^*$. Administrators can use these sets to harden their systems. Thus, they will get maximal security from the configuration hardening and keep the system running.

We showed how we could use existing techniques from the software testing domain to solve a problem in configuration hardening. Since we published our code, we hope administrators can use it to harden their systems. Furthermore, other researchers can improve our approach or the implementation to devise more efficient ways to harden a system without breaking its functionality.

However, as we have discussed in section VIII, administrators need automatic tests to apply our approach. Thus, we advocate for more automatic testing. Only if administrators have sufficient automatic tests to ensure that all system functions are still working, they will have the courage to implement security measures of any kind.

Until administrators have automatic testing, they can test the covering arrays of the guides in A/B tests: Then, they apply one tuple to the machines of selected employees. If employees cannot do their work due to the applied rules, they report this to the administrator. The administrator marks the tuple as breaking and reverts the rules of the tuple on the employee's machine so they can work again. After the administrator has tested all tuples, they can use our tool to find the breaking rules based on the breaking tuples. Ultimately, we will need more testing to have more secure systems in the future.

References

[1] IBM Corporation. (2022, 7) Cost of a Data Breach Report 2022. [Online]. Available: https://www.ibm.com/downloads/cas/3R8N1DZJ

[2] C. Dietrich, K. Krombholz, K. Borgolte, and T. Fiebig, "Investigating System Operators' Perspective on Security Misconfigurations," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: ACM, 2018, pp. 1272–1289. [Online]. Available: http://doi.acm.org/10.1145/3243734.3243794

[3] P. Stöckle, B. Grobauer, and A. Pretschner, "Automated Implementation of Windows-Related Security-Configuration Guides," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '20. New York, NY, USA: Association for Computing Machinery, 2020, pp. 598–610. [Online]. Available: https://doi.org/10.1145/3324884.3416540

[4] D. Kuhn, D. Wallace, and A. Gallo, "Software fault interactions and implications for software testing," *IEEE Transactions on Software Engineering*, vol. 30, no. 6, pp. 418–421, 2004.

[5] R. Kuhn, R. Kacker, Y. Lei, and J. Hunter, "Combinatorial Software Testing," *Computer*, vol. 42, no. 8, pp. 94–96, 2009.

[6] Y. Lei, R. Kacker, D. R. Kuhn, V. Okun, and J. Lawrence, "IPOG: A General Strategy for T-Way Software Testing," in *14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS'07)*, 2007, pp. 549–556.

[7] ——, "IPOG/IPOG-D: efficient test generation for multi-way combinatorial testing," *Software Testing, Verification and Reliability*, vol. 18, no. 3, pp. 125–148, 2008. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.381

[8] L. Yu, Y. Lei, R. N. Kacker, and D. R. Kuhn, "ACTS: A Combinatorial Test Generation Tool," in *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, 2013, pp. 370–375.

[9] P. Stöckle, I. Pruteanu, B. Grobauer, and A. Pretschner, "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*, ser. CODASPY '22. New York, NY, USA: Association for Computing Machinery, 2022, pp. 137–142. [Online]. Available: https://doi.org/10.1145/3508398.3511525

[10] A. Pretschner, "Defect-Based Testing," *Dependable Software Systems Engineering*, vol. 84, 2015.

[11] C. Yilmaz, M. Cohen, and A. Porter, "Covering arrays for efficient fault characterization in complex configuration spaces," *IEEE Transactions on Software Engineering*, vol. 32, no. 1, pp. 20–34, 2006.

[12] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine Learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[13] R. K. Brayton, G. D. Hachtel, C. T. McMullen, and A. L. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic Publishers, 1984.

[14] C. Drake, "PyEDA: Data Structures and Algorithms for Electronic Design Automation," in *Proceedings of the 14th Python in Science Conference*, ser. SCIPY 2015, 2015, pp. 25–30.

[15] C. for Internet Security. (2022) CIS Benchmark for Windows 10, version 1909. [Online]. Available: https://www.cisecurity.org/benchmark/microsoft%5Fwindows%5Fdesktop

[16] M. Velez, P. Jamshidi, N. Siegmund, S. Apel, and C. Kästner, "On Debugging the Performance of Configurable Software Systems: Developer Needs and Tailored Tool Support," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22. New York, NY, USA: Association for Computing Machinery, 2022, pp. 1571–1583. [Online]. Available: https://doi.org/10.1145/3510003.3510043

[17] C. Dubslaff, K. Weis, C. Baier, and S. Apel, "Causality in Configurable Software Systems," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22. New York, NY, USA: Association for Computing Machinery, 2022, pp. 325–337. [Online]. Available: https://doi.org/10.1145/3510003.3510200

[18] G. A. Randrianaina, X. Tërnava, D. E. Khelladi, and M. Acher, "On the Benefits and Limits of Incremental Build of Software Configurations: An Exploratory Study," in *ICSE 2022 - 44th International Conference on Software Engineering*, Pittsburgh, Pennsylvania / Virtual, United States, May 2022, pp. 1–12. [Online]. Available: https://hal.archives-ouvertes.fr/hal-03547219

[19] M. Ul Haque, M. M. Kholoosi, and M. A. Babar, "KGSecConfig: A Knowledge Graph Based Approach for Secured Container Orchestrator Configuration," in *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2022, pp. 420–431.

[20] P. Stöckle, T. Wasserer, B. Grobauer, and A. Pretschner, "Automated Identification of Security-Relevant Configuration Settings Using NLP," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '22, IEEE/ACM. Rochester, MI, USA: Association for Computing Machinery, 2022. [Online]. Available: https://mediatum.ub.tum.de/doc/1685448/1685448.pdf

[21] R. Mandl, "Orthogonal Latin Squares: An Application of Experiment Design to Compiler Testing," *Commun. ACM*, vol. 28, no. 10, pp. 1054–1058, oct 1985. [Online]. Available: https://doi.org/10.1145/4372.4375

[22] H. Wu, C. Nie, J. Petke, Y. Jia, and M. Harman, "An Empirical Comparison of Combinatorial Testing, Random Testing and Adaptive Random Testing," *IEEE Transactions on Software Engineering*, vol. 46, no. 3, pp. 302–320, 2020.

[23] D. R. Kuhn, R. N. Kacker, and Y. Lei, "Practical Combinatorial Testing," National Institute of Standards and Technology, Gaithersburg, MD, Tech. Rep. NIST Special Publication (SP) 800-142, 10 2010.