Sichere Konfigurationshärtung laufender Systeme

Patrick Stöckle
TU München
patrick.stoeckle@tum.de

Michael Sammereier
TU München
michael.sammereier@tum.de

Bernd Grobauer
Siemens AG
bernd.grobauer@siemens.com

Alexander Pretschner
TU München
alexander.pretschner@tum.de

Zusammenfassung

Unsichere Standardwerte in Softwareeinstellungen können von Angreifern ausgenutzt werden, um das System, auf dem die Software läuft, zu kompromittieren. Als Gegenmaßnahme gibt es Richtlinien für die Sicherheitskonfiguration, die detailliert angeben, welche Werte für welche Einstellung sicher sind. Die meisten Administratoren sehen jedoch immer noch davon ab, bestehende Systeme zu härten, weil sie befürchten, dass sich die Systemfunktionalität verschlechtert, wenn sichere Einstellungen verwendet werden. Um die Anwendung von Sicherheitskonfigurationsrichtlinien zu fördern, müssen diejenigen Regeln identifiziert werden, die die Funktionalität einschränken würden.

In diesem Artikel wird unser Ansatz vorgestellt, mithilfe von Combinatorial Testing problematische Regelkombinationen zu finden und mit Techniken des maschinellen Lernens die problematischen Regeln innerhalb dieser Kombinationen zu identifizieren. Die Administratoren können dann nur die unproblematischen Regeln anwenden und so die Sicherheit des Systems erhöhen, ohne dessen Funktionalität zu beeinträchtigen. Um die Nützlichkeit unseres Ansatzes zu demonstrieren, haben wir ihn auf reale Probleme aus Gesprächen mit Administratoren bei Siemens angewandt und die problematischen Regeln in diesen Fällen gefunden. Wir hoffen, dass dieser Ansatz mehr Administratoren dazu motiviert, ihre Systeme zu härten und damit die allgemeine Sicherheit ihrer Systeme zu erhöhen.

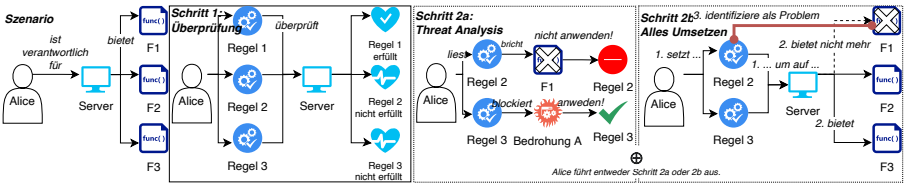


Abbildung 1: Aktuelles Verfahren zur Härtung laufender Systeme

1 Einleitung

Bei 15 % aller Databreaches in der Cloud nutzen die Angreifer Fehlkonfigurationen aus [4]. Dennoch gibt es verschiedene Faktoren - vor allem mangelndes Wissen -, die Administratoren aktuell daran hindern, ihre Systeme sicher zu konfigurieren [2]. Für die meisten Systeme können wir den *Mangel an Wissen* lösen, indem wir eine Richtlinie zur Sicherheitskonfiguration von unabhängigen Organisationen wie dem Center for Internet Security (CIS) verwenden. Eine Richtlinie ist ein Satz von Regeln, und jede Regel gibt an, auf welchen Wert der Administrator eine bestimmte Konfigurationseinstellung setzen muss, um das System sicherer zu machen. Die Administratoren zögern meistens jedoch, diese Konfigurationshärtung auf Systeme in der Produktion anzuwenden. Selbst wenn eine Richtlinie zur Verfügung steht und die notwendigen Prozesse und Tools zur effizienten Implementierung von Leitfäden vorhanden sind [14], werden sie durch die Angst, die bestehende Funktionalität zu zerstören, entmutigt. In diesem Artikel befassen wir uns mit diesem Problem der Härtung bestehender Systeme und der automatischen Erkennung von Regeln, die die Funktionalität einschränken.

1.1 Motivierendes Beispiel

Abbildung 1 zeigt das aktuelle Verfahren beim Sichern laufender Systeme. Administratorin Alice ist für einen Server verantwortlich, auf dem wesentliche Geschäftsfunktionen ausgeführt werden (s. Abbildung 1, *Szenario*). In der Praxis überprüfen wir, ob diese Funktionen noch vorhanden sind, mit automatischen Tests auf verschiedenen Abstraktionsebenen, von Unit-Tests bis hin zu End-to-End-Tests. Alice möchte den Server sicher konfigurieren und verwendet eine CIS-Richtlinie. Diese Richtlinie enthält mehr als 500 Regeln. Alice prüft automatisch, wie viele Regeln der Richtlinie das System derzeit nicht erfüllt (s. Abbildung 1, *Schritt 1*).

Eine aktuelle Studie hat gezeigt, dass ein System, das Windows 10 oder Microsoft Office in der Standardkonfiguration verwendet, im Durchschnitt nur $\approx 17,7\%$ der entsprechenden CIS-Regeln [13] erfüllt. Somit werden bei den Prüfungen im Durchschnitt mehr als 410 nicht konforme Regeln gemeldet werden. Alice könnte diese Regeln durchgehen und für jede Regel zwei Entscheidungen treffen (s. Abbildung 1, *Schritt 2a*): Erstens: Ist diese Regel wichtig für die Sicherheit ihres Servers? Eine bestimmte Regel könnte allgemein von Vorteil sein, aber die Bedrohung, auf die die Regel abzielt, könnte für ihren Server nicht relevant sein. Zweitens: Könnte die Regel das aktuelle Verhalten des Systems beeinträchtigen, d. h., schaltet die Regel eine Funktion aus, die die vorhandenen Systeme auf dem Server noch benötigen?

Beide Entscheidungen sind komplex und zeitaufwändig. Für die erste benötigen wir eine Threat-Analysis, um die relevanten Assets und die daraus resultierenden Bedrohungen zu ermitteln. Für die zweite Entscheidung müssen wir die potenziellen Nebenwirkungen der Regeln kennen. Für jede Nebenwirkung müssen wir dann prüfen, ob sie die auf dem Server laufende Software beeinträchtigt. Obwohl die Regeln eine Beschreibung der potenziellen Nebenwirkung enthalten, muss Alice wissen, wie die vorhandene Software funktioniert, um potenzielle Probleme mit den anzuwendenden Regeln abzuschätzen. Wenn wir mit einer Minute pro Entscheidung rechnen, würde der Prozess mehr als sechs Stunden dauern. Es gibt nur wenige Systeme, bei denen eine so umfangreiche Analyse wirtschaftlich sinnvoll ist. Alice hat also zwei Möglichkeiten. Entweder sie setzt **alle** nicht erfüllten Regeln auf dem Server um oder sie härtet den Server gar nicht ab.

Wenn sie sich für die Anwendung aller nicht konformen Regeln entscheidet, werden höchstwahrscheinlich Probleme mit der bestehenden Funktionalität auftreten, die durch erneute Regressionstests nach dem Härtungsprozess aufgedeckt werden. Für unser Beispiel nehmen wir also an, dass es solche **gebrochene** Funktionalitäten gibt. Wenn Alice die Software kennt, kann sie darauf schließen, welche Regeln die Probleme verursachen könnten (s. Abbildung 1, *Schritt 2b*); wir nennen diese Regeln funktions-brechende oder einfach **brechende** Regeln. Andernfalls muss sie so lange Regeln deaktivieren, bis sie alle Regeln gefunden hat, die die Funktionalität brechen.

Wenn Alice alle Regeln findet, die die Funktionalität einschränken, und alle anderen Regeln anwendet, kann sie die Funktionalität des Systems garantieren und die durch die Konfigurationshärtung gewonnene Sicherheit maximieren. Normalerweise weiß Alice nicht, wie viele Regeln sie ausschließen muss und

wie die Regeln zusammenwirken. Mehrere Regeln können dieselbe Funktionalität verletzen, also muss sie alle ausschließen. Darüber hinaus muss sie möglicherweise nicht alle Regeln entfernen, da die Kombination mehrerer Regeln eine Funktionalität bricht, sodass sie nur eine der entsprechenden Regeln ausschließt.

Bei der Anwendung einer CIS-Richtlinie auf ein Testsystem bei Siemens mussten wir 9 der 500 Regeln ausschließen, d. h. wenn wir die Anzahl der auszuschließenden Regeln wüssten, hätten wir nur $\binom{500}{9}$ Kandidaten. In der Praxis brauchen wir längst nicht so viele Kandidaten zu untersuchen, aber im Durchschnitt kostet dies viel mehr Zeit als der Entscheidungsprozess, was diesen Ansatz noch kostspieliger macht als den vorherigen.

Daher wird Alice nur Regeln implementieren, bei denen sie sich zu 100 % sicher ist, dass sie keine Funktion brechen oder das System überhaupt nicht härten. Dieses Verhalten, dem Risiko von Problemen mit der Softwarefunktionalität durch Vernachlässigung der Sicherheitskonfiguration auszuweichen, ist weit verbreitet. Wir haben eine Fallstudie mit zwei Kommunen in Süddeutschland durchgeführt: Obwohl es für diese Kommunen eine Richtlinie gab, erfüllten ihre Systeme nur 12 % bzw. 35 % der Regeln. Die Administratoren argumentierten, dass sie eine sehr heterogene Umgebung und hohe Verfügbarkeitsanforderungen haben und daher nicht an der Funktionalität des Systems rütteln wollten. Man kann dies als anekdotischen Beweis ansehen, aber wir haben die gleiche Argumentation von Administratoren an der Technischen Universität München gehört.

1.2 Problem

Das Hauptproblem, das in diesem Artikel behandelt wird, ist das folgende: Wir wollen ein bestehendes System im Bezug auf seine Konfigurationseinstellungen abhärten, ohne seine aktuelle Funktionalität zu beeinträchtigen. Daher wollen wir für eine gegebene Richtlinie, ein gegebenes System und seine gegebene Funktionalität eine maximale Teilmenge von Regeln finden, die die Funktionalität des Systems nicht beeinträchtigt.

1.3 Lösungsansatz

Um dieses Problem zu lösen, verwenden wir bestehende Verfahren aus dem Combinatorial Testing in Kombination mit Entscheidungsbäumen, um eine solche maximale Teilmenge effizient zu finden (s. Abbildung 2). Zunächst

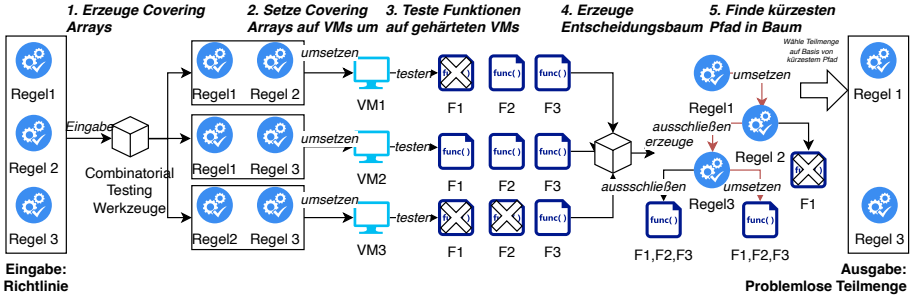


Abbildung 2: Neuer Ansatz zur Identifizierung brechender Regeln

erzeugen wir auf der Grundlage der gegebenen Regeln sogenannte Covering Arrays. Zweitens wenden wir die im aktuellen Array-Eintrag angegebenen Regeln an, testen alle Funktionen und speichern das entsprechende Ergebnis. Drittens trainieren wir Entscheidungsbäume mit den Daten aus dem vorherigen Schritt. Viertens verwenden wir den kürzesten Pfad, der dazu führt, dass alle Funktionen funktionieren, um eine Menge von Unterbrechungsregeln zu finden, die uns zu einer maximalen Teilmenge führt, die die Funktionalität nicht unterbricht.

1.4 Forschungsbeitrag

Unser Beitrag besteht darin, dass wir bewährte Techniken aus dem Software-Engineering, genauer gesagt aus dem Bereich des Software Testings, auf den Bereich der Sicherheitskonfiguration übertragen, um ein weit verbreitetes Problem zu lösen. Außerdem planen wir unseren Code zur Verfügung zu stellen, damit Praktiker ihn nutzen können, um brechende Regeln zu finden.

2 Ansatz

2.1 Erzeuge Covering Arrays aus Richtlinien

Ein naiver Lösungsansatz wäre jede mögliche Kombination von Regeln zu testen und anschließend die Kombinationen ohne fehlgeschlagene Tests nach der Menge mit den meisten angewandten Regeln zu durchsuchen; dies ist aber völlig ineffizient. Im Software Testing haben Forscher bereits ein ähnliches Problem gelöst: Wenn wir ein Programm mit vielen verschiedenen Parametern testen wollen, wollen wir es in allen möglichen Kombinationen der Parameter testen.

Mithilfe des Combinatorial Testings können wir Programme testen, ohne alle Parameterkombinationen zu probieren [5]. Mit Combinatorial Testing können wir je nach gewünschter *Stärke* die zu testenden Kombinationen verglichen mit dem Testen aller Kombinationen drastisch reduzieren. Allerdings können wir nur Fehler bis zu dieser Stärke zuverlässig erkennen, d. h., Combinatorial Testing der Stärke 2 erkennt zuverlässig alle Fehler, die durch die Kombination von zwei oder weniger Parametern verursacht werden [7].

Wenn wir Combinatorial Testing verwenden, müssen wir zuerst die gewünschte Stärke bestimmen. Da es keine Daten aus dem Bereich der Sicherheitskonfiguration gibt, müssen wir uns auf Daten aus dem Software Testing stützen. Da die Studie von Kuhn et al. keine fehlgeschlagenen Tests mit einer Kombination von mehr als sechs Parametern fand [5], gehen wir davon aus, dass es auch keine Kombination von mehr als sechs Regeln gibt, die eine Funktion zum Scheitern bringt.

Um Combinatorial Testing für die Sicherheitskonfiguration anzuwenden, generieren wir einmal für jede Richtlinie mit Regeln eine Menge von -Tupeln mit `true` oder `false`; `true` an der Position eines Tupels bedeutet, dass wir die Regel in dieser Kombination anwenden. Ein solches Tupel heißt Covering Array. Wir können diese Covering Arrays später für mehrere Systeme mit unterschiedlichen Funktionalitäten wiederverwenden, um brechende Regeln zu finden, nur wenn wir neue Regeln hinzufügen oder entfernen, müssen wir die Covering Arrays neu generieren. Da die Richtlinien Hunderte von Regeln enthalten, benötigten wir Algorithmen, die mit Tupeln dieser Größe umgehen können. Daher verwenden wir die Algorithmen IPOG [8] und IPOG-D [9] und ihre Implementierungen im ACTS-Tool [21], um die Kombinationen zu erzeugen.

Zunächst übersetzen wir die Regeln einer Richtlinie aus ihrem ursprünglichen Format in eine ACTS-Eingabedatei, in der die verwendeten Parameter definiert sind. In dieser Eingabedatei hat jeder Parameter einen Namen und einen Datentyp, d.h. in unserem Szenario die ID der Regel, z.B. `R1_1_1`, und `boolean`. Je nach dem gewählten Grad von Covering Arrays und dem Algorithmus erzeugt ACTS nun die Covering Arrays. Lst. 1 zeigt eine Beispielausgabe. Anschließend übersetzen wir die ACTS-Ausgabe in JSON-Dateien, die wir für die automatische Implementierung von Richtlinien [13] verwenden. Nach diesem ersten Schritt haben wir nun die verschiedenen Covering Arrays in einer Form, die wir automatisch auf einem System implementieren können, um anschließend zu testen, ob und welche Funktionen eingeschränkt wurden.

```
# 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 1: ACTS Ausgabe für eine CIS Richtlinie.

2.2 Teste Funktionen auf gehärteten Instanzen

Je nach Stärke der Covering Arrays generieren wir zwischen 20 und 5545 Tupel für den CIS Windows 10 Guide. Im nächsten Schritt wenden wir jedes Tupel von Regeln an und prüfen mit den vorgegebenen Tests, ob alle Funktionen noch aktiv sind. Wenn ein Test fehlschlägt, wird dies in einer Protokolldatei festgehalten. Nachdem wir diese Prozedur für jedes Tupel ausgeführt haben, sammeln wir die verschiedenen Protokolldateien und führen sie in einer Datei zusammen. Die resultierende Datei gibt an, bei welchem Tupel ein Test fehlgeschlagen ist und bei welchem nicht.

Was sich in der Theorie einfach anhört, war in der Praxis sehr mühsam. Das erste Problem war, dass wir eine Umgebung vorbereiten mussten, in der wir die Software einrichten, alle Regeln in einem Tupel anwenden, die Funktionalitäten testen und das Ergebnis aufzeichnen konnten. Um dieses Problem zu lösen, verwenden die Sicherheitsexperten bei Siemens eine Toolchain mit Ansible, Vagrant und AWS, um mehrere virtuelle Maschinen (VM) effizient bereitzustellen, die sie unabhängig und parallel konfigurieren können. Eine Alternative ist die Verwendung von Vagrant und einem Hypervisor wie VirtualBox, um die VMs lokal auszuführen. Das zweite Problem waren Reihenfolgeneffekte, d. h., ein Test schlägt nicht wegen des aktuell angewendeten Tupels fehl, sondern wegen des vorherigen. Um diese Effekte zu vermeiden, könnten wir entweder die VM nach jedem Testlauf komplett zurücksetzen oder nur einen Soft-Reset durchführen, bei dem wir nur die angewandten Regeln zurücksetzen. Das komplette Zurücksetzen kostet mehr Zeit als der Soft-Reset, aber es besteht kein Risiko von Nebeneffekten. Mit unserer Implementierung können wir für jedes Tupel eine VM-Instanz erzeugen oder wir erzeugen mehrere Instanzen und verteilen die Tupel gleichmäßig auf die Instanzen. Das dritte Problem waren die automa-

```
[{ "name": "custom_1", "breaking": true,
   "rules": ["R1_1_1", "R1_2_2", "R1_3_1", "..."]},
{  "name": "custom_2", "breaking": false,
   "rules": ["R1_1_2", "R1_1_4", "R1_1_7", "..."]},
"..."]
```

Listing 2: Beispielergebnis des Testprozesses.

tischen Tests. Idealerweise würden wir Tests aus der Industrie verwenden, um die Funktionalität zu testen. Wie wir bereits erwähnt haben, sichern jedoch nur wenige Unternehmen ihre Systeme durch sichere Konfiguration. Diese Unternehmen prüfen manuell, ob ihre Systeme nach Härtung noch funktionieren und wir mussten daher unsere automatischen Tests selbst erstellen. Wenn jedoch alle Tests bestanden werden, obwohl die Regeln die Funktionalität brechen, wird unser Testprozess dies nicht feststellen und wir die Probleme erst auf den produktiven Systemen bemerken. Es ist daher entscheidend, geeignete Tests zu finden.

Nachdem wir diese Probleme gelöst haben, besteht unser Testverfahren aus folgenden Schritten:

1. Zunächst bereiten wir ein Image mit unserer Software und allen benötigten Abhängigkeiten vor. Vagrant verwendet diese Box, um die verschiedenen VM-Instanzen einzurichten.

2. Wir bereiten für jede VM-Instanz ein Verzeichnis mit der auszuführenden Software, den Tests, der Richtlinie und den zu testenden Covering Arrays vor. Wenn wir mehrere Arrays auf einer Instanz testen wollen, verteilen wir sie gleichmäßig.

3. Wir starten die Instanzen entweder parallel oder sequentiell.

4. Wir führen alle Tests auf einer beliebigen Instanz aus, um zu sehen, ob die Funktionalität in der Standardkonfiguration funktioniert. Wenn die Tests fehlschlagen, bevor Regeln angewendet werden, dann muss es ein Problem in den Tests oder im Aufbau geben, das wir beheben müssen.

5. Wir setzen automatisiert alle Regeln auf einer Instanz um.

6. Wenn die Richtlinie eine brechende Regel enthält, wird mindestens ein Test fehlschlagen, d.h., wenn keine Tests fehlschlagen, können wir alle Regeln sicher anwenden und den Prozess an diesem Punkt beenden.

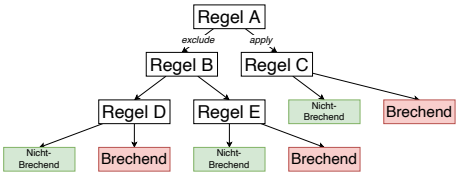


Abbildung 3: Vereinfachtes Beispiel eines generierten Entscheidungsbaumes.

7. Wenn wir den Soft-Reset verwenden, muss der Umkehrmechanismus funktionieren. Es gibt jedoch einige Regeln, die wir aus technischen Gründen nicht umkehren können. Daher versuchen wir, alle Regeln umzukehren und die Tests erneut auszuführen. Wenn einige Tests immer noch fehlschlagen, gibt es ein Problem mit dem Umkehrmechanismus.

8. Wir nehmen das erste ungetestete Covering Array und setzen alle Regeln aus diesem auf einer Instanz um.

9. Wir führen die Tests aus und speichern in einer JSON-Datei, ob es fehlgeschlagene Tests gab.

10. Wir führen einen Soft- oder Hard-Reset durch, um Reihenfolgeneffekte zu vermeiden. Danach gehen wir zurück zu Schritt 7, bis wir alle Covering Arrays angewendet haben.

11. Sammeln der Ergebnisse: Nachdem wir alle Covering Arrays angewendet und getestet haben, sammeln wir alle Ergebnisse aus den verschiedenen Instanzen und kombinieren sie zu einer einzigen JSON-Datei; Lst. 2 zeigt eine solche JSON-Datei.

12. Wir löschen die Instanzen.

Wir haben nun alle Covering Arrays getestet, und das resultierende JSON gibt an, welche Covering Arrays welche Tests fehlschlagen lassen. Im nächsten Schritt verwenden wir diese Informationen, um abzuleiten, welche Regeln die Fehlschläge verursacht haben.

2.3 Analyse der Testergebnisse

Als Nächstes analysieren wir diese Daten, um eine maximale Menge an nichtbrechenden Regeln zu bestimmen. Wir könnten natürlich auch die Covering

Arrays, die zu Fehlern geführt haben, an die Administratoren weitergeben, damit diese ihre Systeme so anpassen, sodass sie auch dann funktionieren, wenn diese Covering Arrays angewendet wurden. Insbesondere bei Altsystemen kann es jedoch schwierig sein, Änderungen an der Software oder dem Gesamtsystem vorzunehmen. Daher wollen wir für diesen Artikel die Richtlinie anpassen, indem wir potenziell problematische Regeln für unser System entfernen, und nicht das System an sich ändern. In der Praxis bei Siemens würden allerdings die Administratoren entscheiden, ob sie andere Sicherheitsmaßnahmen benötigen, um den aus den ausgeschlossenen Regeln resultierenden Risiken vorzubeugen.

Wir haben den von Yilmaz et al. [20] beschriebenen Ansatz für unser Szenario angepasst. Sie verwendeten maschinelles Lernen, um die Parameter zu finden, die zum Scheitern eines Tests führen, und trainierten einen Entscheidungsbaum auf ihre Testergebnisse. Ein Beispiel für einen Entscheidungsbaum ist in Abbildung 3 zu sehen. Die Knoten stellen Regeln aus den Covering Arrays dar. Der Algorithmus berechnet verschiedene Partitionen von brechenden und nicht-brechenden Covering Arrays, indem er prüft, ob wir eine Regel anwenden oder nicht. In Abbildung 3 unterscheidet der Algorithmus zunächst zwischen Covering Arrays, auf die *Regel A* angewendet wird oder nicht. Jeder Blattknoten gibt an, ob die Funktionalität gebrochen wurde oder nicht, wenn wir die Regeln auf dem Pfad zwischen der Wurzel und dem Blattknoten anwenden beziehungsweise ausschließen. In Abbildung 3 führt die Anwendung von *Regel A*, aber der Ausschluss von *Regel C* zu einem nicht-brechenden Blatt, d.h. *Regel C* ist eine brechende Regel.

Wir könnten diesen Entscheidungsbaum verwenden, um vorherzusagen, ob ein Covering Array, das wir noch nicht getestet haben, zu Problemen führt oder nicht. Für unseren Anwendungsfall sind wir jedoch nur an den nicht-brechenden Blättern mit den wenigsten ausgeschlossenen Regeln interessiert. Daher verwenden wir Kürzeste-Wege-Algorithmen wie z.B. den Dijkstra-Algorithmus, um diese Blätter zu finden. Wir geben den Kanten, die eine Regel anwenden, d. h. den rechten Kanten in Abbildung 3, den Wert 0 und allen anderen Kanten den Wert 1. Die Gesamtkosten eines Pfades von der Wurzel zu einem Blatt sind also die Anzahl der Regeln, die wir nicht angewendet haben. Der kürzeste Pfad, der zu einem nicht-brechenden Blatt führt, ist wiederum der Pfad mit der geringsten Anzahl von nicht angewandten Regeln, d.h. eine maximale nicht-brechende Menge an Regeln. In unserem Beispielbaum erreichen wir ein nicht-brechendes Blatt, indem wir *Regel A* anwenden und *Regel C* ausschließen. Damit enthält unsere maximale nicht-brechende Menge für das Beispiel alle Regeln außer *C*.

Wir haben unseren Ansatz mit *scikit-learn* [11] implementiert. Zunächst übersetzen wir die Daten aus dem vorherigen Schritt in das Scikit-Format. Zweitens trainieren wir einen Entscheidungsbaum auf den Daten. Drittens fügen wir die Gewichte zu dem gelernten Entscheidungsbaum hinzu. Viertens führen wir einen Algorithmus für den kürzesten Weg auf dem gewichteten Entscheidungsbaum aus, um ein nicht-unterbrechendes Blatt zu finden. Fünftens: Wir folgen dem Pfad von der Wurzel zu diesem Blatt. Wenn die aktuelle Kante das Gewicht 1 hat, entfernen wir die aktuelle Regel aus der Richtlinie. Am Ende dieses Schrittes ist die resultierende Menge eine maximale nicht-brechende Menge.

3 Evaluation

Bei der Evaluierung unseres Ansatzes und seiner Umsetzung konzentrieren wir uns auf die folgenden Forschungsfragen:

RQ1 Können wir eine maximale, nicht-brechende Teilmenge einer Richtlinie in Bezug auf gegebene Funktionalitäten durch Combinatorial Testing finden? Welche brechenden Kombinationen können wir feststellen? Theoretisch sollte die Stärke der Covering Arrays die Obergrenze für den Grad der erkannten Kombinationen darstellen. Allerdings könnte die Kombination mit den Entscheidungsbäumen die Leistungsfähigkeit unseres Ansatzes in der Praxis verringern.

RQ2 Wie viel Zeit benötigt unser Ansatz? Ist dies ein angemessener Zeitaufwand für die Härtung eines Systems?

Wir haben die CIS-Richtlinie für Windows 10, Version 1909 [1] mit 507 Regeln für unsere Evaluation verwendet. Um die Regeln automatisch anzuwenden und zurückzusetzen, haben wir die Richtlinie in das Scapolite-Format [13] umgewandelt. Im Folgenden erörtern wir, wie wir die verschiedenen Schritte unseres Ansatzes evaluiert haben.

3.1 Generierung

Zunächst evaluieren wir die Generierung der Covering Arrays aus einer bestehender Richtlinie. Um **RQ1** zu beantworten, wollten wir wie in Unterabschnitt 2.1 beschrieben Covering Arrays mit einer Stärke bis zu 6 sowohl mit

```
[["R_A", "R_B"], ["R_C", "R_D"]]
```

Listing 3: Beispiel für die Definition einer brechenden Kombination.

IPOG als auch mit IPOG-D erzeugen. Wir haben das ACTS-Tool auf einem Server mit zwei Intel Xeon E5-2687W v3-CPUs mit 40 Kernen und insgesamt 500 GB RAM laufen lassen, von dem wir bis zu 340 GB genutzt haben.

3.2 Testing

Wir haben unser Testverfahren in drei Teilen evaluiert. Im ersten Teil simulierten wir, welche Regeln die Funktionen brechen würden. Hier haben wir zunächst 51 Kombinationen von Regeln definiert, die die Funktionalität brechen; eine Kombination ist die leere Menge, d.h. der Spezialfall, in dem wir unsere Richtlinie ohne Probleme anwenden können. Diese Kombinationen haben wir als logische Formeln in disjunktiver Normalform definiert. Lst. 3 zeigt, wie wir die Formel $(R\_A \wedge R\_B) \vee (R\_C \wedge R\_D)$ ausdrücken. Wenn wir alle Regeln der ersten oder der zweiten Teilformel anwenden, wird die Funktionalität des Systems unterbrochen, z. B. können wir `R_A` und `R_B` anwenden, um die Funktion zu unterbrechen, aber nicht `R_A` und `R_C`. Darüber hinaus haben wir für jede der nicht leeren Kombinationen drei zusätzliche, zufällige Varianten getestet, d.h. wir haben die IDs der Regeln durch zufällige andere IDs ersetzt, um mögliche Nebeneffekte aufgrund der Reihenfolge der Regeln zu vermeiden. Insgesamt haben wir also 201 Kombinationen getestet.

Für jede der Kombinationen sind wir anschließend die Covering Arrays durchgegangen und haben für jedes Covering Array die Ergebnisdatei erstellt: Wenn ein Covering Array die Bedingungen einer brechenden Kombination erfüllt, markieren wir das Covering Array als brechend, sonst als nicht-brechend. Anschließend kombinieren wir alle Ergebnisdateien und analysieren sie mit unserem Analyse-Ansatz. Wir haben für die Covering Arrays beide Algorithmen mit unterschiedlichen Stärken verwendet.

Im zweiten Teil haben wir den gesamten Prozess evaluiert, aber mithilfe von generierten Mock-Tests, die auf den definierten Kombinationen basieren. Wie in Unterabschnitt 2.2 beschrieben, führen wir für jede brechende Kombination alle Schritte aus. Bei den Tests in Schritt 8 wird jedoch geprüft, ob das aktuelle System die aktuelle Kombination von brechenden Regeln erfüllt. Ist dies der Fall, wird das aktuelle Covering Array als fehlgeschlagen markiert. Mit diesen

generierten Mock-Tests können wir viele Kombinationen effizient testen. Im Gegensatz zur Simulation erforderte dieser Schritt deutlich mehr Zeit. Die Ausführung der Mock-Tests ist zwar schnell, aber das Einrichten der VM und das Umsetzen und Zurücknehmen der Regeln nimmt einige Zeit in Anspruch. Wir haben diesen Teil der Evaluation mit den vom IPOG-D generierten Covering Arrays der Stärke 4 auf zwei VMs durchgeführt.

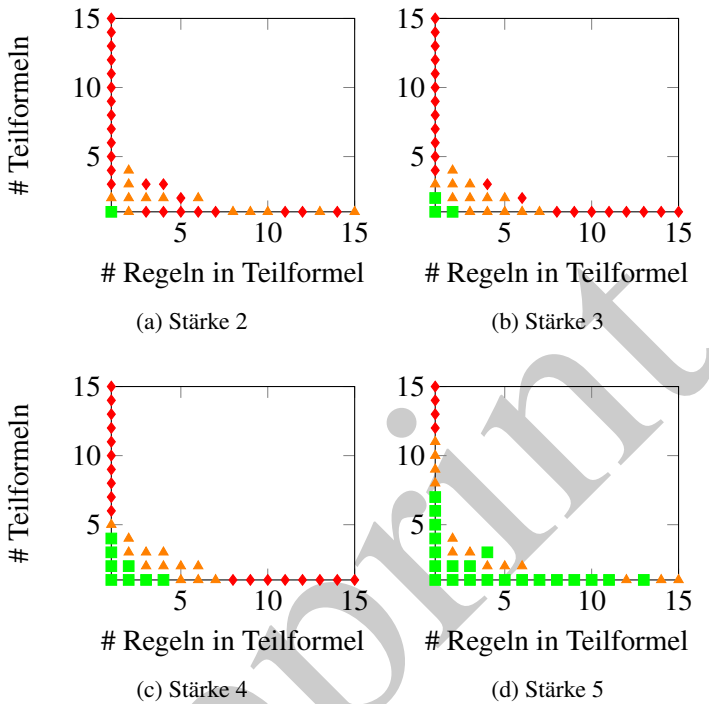
Im dritten Teil haben wir unser Verfahren mit einem realen Test evaluiert. Hier brauchten wir ein kleines Programm, das fehlschlägt, wenn die gesamte Richtlinie angewendet wird. Wir haben uns für ein einfaches PowerShell-Skript entschieden, das einen neuen Benutzer mit einem Kennwort anlegt und den Benutzer anschließend wieder löscht. Dieser Test symbolisiert den Albtraum aller Administratoren wie Alice: Es ist eine einfache Aufgabe, die in der Regel nur Sekunden dauert, aber nach der Härtung nicht mehr funktioniert. Also müssen die Administratoren nun stundenlang nachforschen, welche Regel die Funktionalität gebrochen hat. Als Nächstes führten wir alle Schritte unseres Prozesses mit zwei parallel laufenden Instanzen durch und markierten die Tupel als fehlgeschlagen, wenn das Skript fehlschlug. Wiederum gaben wir das Ergebnis an unsere Analysekomponenten, um die maximale Menge an nicht-brechenden Regeln zu finden. Schließlich wendeten wir diese Menge an und prüften, ob das Skript noch funktionierte. Auch diesen Teil der Auswertung haben wir mit den vom IPOG-D erzeugten abdeckenden Arrays der Stärke 4 auf zwei VMs durchgeführt.

Während uns die Simulation half, **RQ1** zu beantworten, haben wir in Schritt 2 und Schritt 3 die benötigte Zeit gemessen, da dies wesentlich zur Beantwortung von **RQ2** betrug.

3.3 Analyse

Wir brauchen die Analyse der Testergebnisse, um die allgemeine Qualität unseres Ansatzes zu beurteilen. Wir haben aber auch verschiedene Faktoren verglichen: Erstens haben wir den Einfluss verschiedener Parameter, z.B. für die Entscheidungsbaumgenerierung, verglichen. Zweitens, haben wir verschiedene Algorithmen, die eine optimale Lösung im Baum suchen, verglichen.

Wir bewerteten für die verschiedenen Varianten, ob sie eine korrekte maximale nicht-unterbrechende Menge berechneten, und prüften, wie stark sie sich im Falle einer falschen Ausgabe unterscheiden. So können wir **RQ1** beantworten. Außerdem haben wir gemessen, wie lange die Berechnung in Abhängigkeit von der Eingabe dauert, was zur Beantwortung von **RQ2** beiträgt.



(a) Stärke 2 (b) Stärke 3

(c) Stärke 4 (d) Stärke 5

Abbildung 4: Verteilung der korrekten Ergebnisse im Testset.

4 Ergebnisse

Tabelle 1 zeigt für IPOG und IPOG-D die Anzahl an Einträgen pro Covering Array für verschiedene Stärken. Wie erwartet wächst die Anzahl exponentiell mit zunehmender Stärke. Aufgrund der benötigten Zeit konnten wir jedoch keine Covering Arrays der Stärke 5 mit IPOG oder 6 mit IPOG-D erzeugen. Tabelle 2 zeigt die benötigte Zeit zur Erzeugung der Covering Arrays. Für alle Stärken zwischen 2 und 4 ist die Zeit ein Durchschnitt von 5 Messungen; für 5 wurde die Zeit nur einmal gemessen. IPOG-D ist - wie erwartet - schneller als IPOG und wir konnten damit sogar die Covering Arrays mit der Stärke 5 berechnen, obwohl wir dafür $\approx 13,7$ Tage benötigten. IPOG-D erzeugte jedoch mehr Einträge für Covering Arrays der gleichen Stärke.

Abbildung 4 zeigt unsere Simulationsergebnisse mit Covering Arrays der Stärke 2 bis 5. Auf der x-Achse ist die Anzahl der Klauseln und auf der y-Achse

Algorithmus \ Stärke	2	3	4	5
IPOG-D	20	78	305	5545
IPOG	20	70	209	-

Tabelle 1: Anzahl der erzeugten Einträge je nach verwendetem Algorithmus und Stärke.

die maximale Anzahl von Regeln innerhalb einer Klausel angegeben. Eine rote Raute bedeutet, dass unser Ansatz für keine der Mengen in diesem Cluster die Lösung korrekt identifizieren konnte. Ein oranges Dreieck bedeutet, dass unser Ansatz einige korrekte Lösungen für die Mengen in diesem Cluster korrekt identifiziert hat, aber nicht alle. Ein grünes Quadrat bedeutet, dass unser Ansatz alle Lösungen für diesen Cluster korrekt identifiziert hat. Aufgrund der Stärke der abdeckenden Felder haben wir ein Dreieck aus grünen Quadraten in der unteren linken Ecke erwartet. Unser Ansatz schnitt jedoch bei Covering Arrays der Stärke 2 und 3 schlechter als erwartet ab, bei 5 jedoch besser. Der Hauptgrund für die Anfangs schlechte Performance ist der Mangel an Daten für den Entscheidungsbaum. Der Entscheidungsbaum hat keine Daten zur Partitionierung und kann daher keine auszuschließenden Regeln bestimmen. In einigen Fällen konnte der Algorithmus Teilmengen der richtigen Lösungen finden. Für unsere Evaluation betrachten wir diese Fälle als ungültige Ergebnisse, aber sie können in der Praxis dazu beitragen, die optimale Lösung zu finden. Überraschenderweise konnte unser Ansatz die korrekten Ergebnisse für einzelne Formeln mit bis zu 11 Regeln pro Teilformel auf der Basis der Covering Arrays der Stärke 5 berechnen, obwohl wir nur korrekte Ergebnisse für bis zu 5 erwartet hatten. Wir haben die nicht korrekt berechneten Bruchregelsätze genauer untersucht. Obwohl richtige Lösungen Teil des Baums waren, dominierten falsche Lösungen aufgrund der kürzeren Länge; wahrscheinlich hätten mehr Einträge im Covering Array geholfen. Im Allgemeinen berechnete unser Ansatz die korrekte Lösung für 77% der Bruchregeln in unserer Evaluationsmenge. Man könnte argumentieren, dass dies zu wenig ist, um den Ansatz in der Praxis zu verwenden, aber unser Beispielsatz enthält weitaus kompliziertere Kombinationen als wir in der Realität erwarten. Bei den realistischen Beispielen, die auf der Annahme beruhen, dass nur bis zu 6 Regeln kombiniert zu einem Problem führen, erreichen wir fast 100 %.

Beim zweiten Teil schlugen im Gegensatz zur Simulation die ersten Tests

Algorithmus \ Stärke	2	3	4	5
IPOG-D	0.2	16	8451	1184478
IPOG	0.7	374	179149	-

Tabelle 2: Zeit (in Sekunden) um Covering Arrays zu erzeugen.

fehl, d.h. Schritt 4 des Testprozesses. Der Grund dafür sind die oben erwähnten $\approx 17{,}7\%$ konformen Regeln auf einem System in seiner Standardkonfiguration. Wenn alle Regeln in einer Klausel bereits mit einem Standardsystem konform sind, schlägt der Mock-Test bei Schritt 4 fehl. Daher mussten wir diese Tests auslassen; eine Alternative wäre die Verwendung eines Images gewesen, bei dem alle Einstellungen auf unsichere Werte gesetzt sind, aber wir hielten dies nicht für ein realistisches Szenario. Abgesehen davon führen die generierten Mock-Tests zu denselben Daten wie die Simulation. Allerdings dauerte der gesamte Prozess hier etwa 12 Stunden.

Im dritten Schritt bewerteten wir die praktische Anwendung, indem wir die Funktionalität mit unbekannten Brechungsregeln testeten. Aus den ersten Tests wussten wir, dass die Funktionalität vor der Anwendung von Regeln funktionierte, aber nicht nach deren Anwendung. Insgesamt dauerte der Testprozess 12 Stunden und ergab 156 brechende und 149 nicht brechende Kombinationen. Entscheidungsbäume, die aus diesen Ergebnissen gelernt wurden, besagten, dass man die Regel `R1_1_4` ausschließen sollte. Danach haben wir die gesamte Richtlinie ohne die Regel `R1_1_4` angewendet, führten unseren Test erneut durch und er war erfolgreich. Regel `R1_1_4` lautet: “Stellen Sie sicher, dass die ‘Mindestlänge des Kennworts’ auf ‘14 oder mehr Zeichen’ gesetzt ist”, aber unser Testskript versuchte, ein Kennwort der Länge 6 zu setzen und schlug daher fehl.

Als Nächstes haben wir verschiedene Techniken verglichen, um die optimale Lösung zu finden. Zunächst wählten wir das nicht-unterbrechende Blatt mit der größten Partition anstelle des Blattes mit dem kürzesten Pfad. Wir haben die von IPOG-D erzeugten 5-Wege-Kombinationen verwendet, um die verschiedenen Pfadalgorithmen zu vergleichen. Der modifizierte Ansatz schnitt hier in den gemischten Clustern besser ab.

Neben der Modifizierung des verwendeten Pfadalgorithmus haben wir auch Änderungen an der Generierung des Entscheidungsbaums evaluiert, z. B. die Einführung eines Minimums für die Anzahl der Testfälle in einer Partition oder

die Änderung der Mindestanzahl von Datenpunkten für eine Aufteilung der Partition. Keine dieser Änderungen führte jedoch dazu, dass mehr korrekte Lösungen identifiziert wurden.

4.1 Diskussion

4.1.1 Identifikation brechender Regeln

Die Ergebnisse unserer Evaluierung haben gezeigt, dass man kombinatorische Tests und insbesondere Covering Arrays verwenden kann, um brechende Kombinationen von Regeln zu finden. Darüber hinaus können wir auf maschinellem Lernen basierende Heuristiken verwenden, um eine maximale nicht-brechende Menge innerhalb einer Richtlinie zu finden. Unsere Ergebnisse zeigen jedoch auch, dass es einige Vorbehalte gibt. Erstens konnten wir keine Covering Arrays mit einer Stärke von 6 erzeugen, obwohl wir annehmen, dass dies die notwendige Stärke ist, um alle brechenden Mengen in der Praxis abzudecken. Selbst die Erstellung von Covering Arrays mit einer Stärke von 5 beanspruchte zu viel Speicherplatz (340 GB) für zu lange (13 Tage), sodass dies für öffentliche Richtlinien, z. B. von der CIS, kaum nützlich, und für private Richtlinien, z.B. bei Siemens, überhaupt nicht wirtschaftlich ist. Daher ist es realistischer, 4-fach abdeckende Arrays zu verwenden, die garantieren, dass wir alle Kombinationen von 4 oder weniger Regeln finden werden. Dennoch zeigen die Ergebnisse, dass unser heuristischer Ansatz einige Lösungen für höhere Kombinationen oder zumindest Teilmengen einer optimalen Lösung finden kann, die den Administratoren helfen könnten. Da wir keine Informationen über die Verteilung der brechenden Kombinationen in der Praxis haben, ist die Antwort von **RQ1** zweigeteilt. Wenn die gesamte oder ein Großteil der brechenden Funktionalität in der Praxis aus 4 oder weniger Regeln resultiert, können Covering Arrays und unsere heuristische Analyse eine maximale nicht-brechende Teilmenge zuverlässig identifizieren. Wenn ein erheblicher Teil der brechenden Funktionalität aus 5 oder mehr Regeln resultiert, können Covering Arrays und die heuristische Analyse nur dabei unterstützen, die optimale Lösung zu identifizieren, aber nicht direkt die optimale Lösung finden.

4.1.2 Aufwand

Die Generierung verschiedener Covering Arrays hängt von der Anzahl der Regeln in der gewählten Richtlinie und der gewünschten Stärke der Kombi-

nationen ab. Dies kann einige Stunden, Tage oder sogar Wochen in Anspruch nehmen. Wir brauchen diese Generierung jedoch nur einmal, wenn der Herausgeber die Richtlinie veröffentlicht, und nicht für jedes System, das wir mit dem gegebenen Richtlinie härten wollen. Die CIS aktualisiert ihre Leitfäden ebenfalls, aber wenn sich die Regeln ändern, müssen sie die Covering Arrays nicht neu generieren. Wenn sie neue Regeln hinzufügen, könnten sie die vorhandenen Felder wiederverwenden, um die Erstellung zu beschleunigen. Wie bereits erwähnt, ist es realistischer, Covering Arrays der Stärke 4 zu verwenden, da die Stärke 5 zu teuer ist.

Wir können den Aufwand für die Identifizierung einer nicht-brechenden Teilmenge für ein gegebenes System auf Basis der Testschritte (s. Unterabschnitt 2.2) folgendermaßen abschätzen. Das Einrichten einer lokalen VM, z. B. mit Vagrant, kann mehrere Minuten dauern, während eine VM in der Cloud, z. B. auf AWS, viel schneller ist. Die Zeit für die Installation der zu testenden Software hängt von der Komplexität der Software ab. In unserer Bewertung haben wir eine Kernfunktion von Windows getestet und daher keine zusätzliche Software installiert. In unserer früheren Studie [13] haben wir gezeigt, dass wir Windows-Regeln in unter einer 1s pro Regel anwenden können, aber die Zeit auch nicht vernachlässigbar ist. Die Zeit für die Tests hängt von der Komplexität der Tests ab. Das Skript benötigte in unserer Auswertung ein paar Sekunden, aber wenn man komplexe Tests verwendet, kann dies Minuten oder Stunden dauern. Das Zurücksetzen dauert dagegen wieder ungefähr so lang wie die Umsetzung. Die Analyse mit den Entscheidungsbäumen und der Algorithmus für den kürzesten Weg braucht nur ein paar Sekunden mit unserer Heuristik.

Unsere Experimente dauern mit Covering Arrays der Stärke 4 mehr als 12 Stunden und beantworten teilweise **RQ2**. Diese Zeit mag angemessen sein, wenn wir sie nur für Releases verwenden, aber sie ist zu lang, um sie in einem kontinuierlichen Integrationskontext zu verwenden. Wir können hauptsächlich zwei Faktoren in der Gleichung beeinflussen: die Anzahl der Einträge in den Covering Arrays und die Anzahl der VM-Instanzen. Wenn wir die Anzahl der Einträge reduzieren, indem wir Covering Arrays mit geringerer Stärke wählen, werden wir einige komplexe Kombinationen nicht erkennen. Daher erhöhen wir eher die Anzahl der VMs, indem wir mehr Instanzen parallel verwenden, z. B. auf Abruf in der Cloud. Wenn wir 30 On-Demand-Instanzen verwenden, führen wir nur 11 Kombinationen auf jeder VM aus. Im Durchschnitt wendete eine Kombination 240 Regeln an, d. h. das Umsetzen der Richtlinie und Zurücksetzen würde um die 240 Sekunden dauern. Unter der Annahme, dass die

automatischen Tests 2 Minuten dauern, würde der gesamte Prozess 2 Stunden in Anspruch nehmen. Um den Preis dieses Prozesses abzuschätzen, haben wir mit 30 On-Demand-Windows-Instanzen mit jeweils 2 Kernen, 4 GB RAM und 25 GB Speicherplatz auf AWS gerechnet: geschätzte Kosten ca. $3. 2 Stunden sind kurz genug, um diesen Prozess in Regressionstests einzubeziehen, die jede Nacht laufen, und $3 sollten günstig genug sein, um den Prozess für die meisten Systeme auszuführen.

5 Forschungsstand

Forschung zum Konfigurationsmanagement findet schon länger statt, aber es gibt auch neue Erkenntnisse zu diesem Thema [3, 12, 17, 18]. Dietrich et al. haben den besten Überblick über die Probleme von Administratoren bei der sicheren Konfiguration vorgestellt [2]. Die in unserem Artikel vorgestellte Arbeit hängt von unseren früheren Arbeiten im Kontext des Security Content Automation Protocols (SCAP) und der Konfigurationshärtung ab, die andere wesentliche Faktoren wie die *schlechte Herstellerdokumentation* [16] ansprechen. Unser Ansatz benötigt automatisch implementierbare Richtlinien, wie sie in [13] vorgestellt werden. Außerdem verwenden wir die in [14, 15] vorgestellten Techniken z.B. für das Bereitstellen der VMs.

Obwohl sie bereits in den 80er Jahren aufkam [10], ist die Forschung zu Combinatorial Testing immer noch sehr aktiv [19]. Kuhn et al. zeigten, dass man Combinatorial Testing verwenden kann, um zu prüfen, ob die Software mit allen Einstellungen der Software selbst funktioniert, aber auch, ob die Software in jeder möglichen Konfiguration eines Systems funktioniert [6]. Wie bereits erwähnt, verwenden wir die Algorithmen IPOG [8] und IPOG-D [9], um unsere Covering Arrays zu generieren. Der für unsere Arbeit wichtigste Artikel war der Ansatz von Yilmaz et al. zum Auffinden von Fehlern beim Testen von Software mit Combinatorial Testing [20]. Sie schlugen darüber hinaus vor, die Ergebnisse mithilfe von Entscheidungsbäumen weiter zu analysieren. Wir haben ihren Ansatz auf den Bereich der Sicherheitskonfiguration übertragen. Allerdings waren sie nur an einer fehlgeschlagenen Kombination interessiert, während wir an einer maximalen Lösung interessiert sind, d. h. an der Anwendung so vieler Regeln wie möglich.

6 Fazit

Wir zeigen in diesem Artikel, dass man kombinatorische Tests verwenden kann, um Kombinationen von brechenden Regeln zu finden, und Heuristiken basierend auf maschinellem Lernen, um eine maximale nicht-brechende Teilmenge einer Richtlinie zu finden. Administratoren können diese Mengen verwenden, um ihre Systeme zu härten. Auf diese Weise erhalten sie die bestmögliche Sicherheit durch die Konfigurationshärtung und halten das System am Laufen.

Außerdem zeigen wir, wie man bestehende Techniken aus dem Bereich des Softwaretests nutzen kann, um ein Problem der Konfigurationshärtung zu lösen. Allerdings benötigen Administratoren automatische Tests, um unseren Ansatz anzuwenden. Daher plädieren wir für mehr automatische Tests. Nur wenn Administratoren über genügend automatische Tests verfügen, um sicherzustellen, dass alle Systemfunktionen noch funktionieren, werden sie den Mut haben, Sicherheitsmaßnahmen jeglicher Art zu implementieren.

Bis die Administratoren über automatische Tests verfügen, können sie die Covering Arrays der Leitfäden in A/B-Tests testen. Dabei wenden sie einen Eintrag der Covering Arrays auf die Rechner ausgewählter Mitarbeiter an. Wenn die Mitarbeiter aufgrund der angewandten Regeln ihre Arbeit nicht erledigen können, melden sie dies dem Administrator. Der Administrator markiert den Eintrag als fehlerhaft und setzt die Regeln des Eintrags auf dem Rechner des Mitarbeiters zurück, sodass dieser wieder arbeiten kann. Nachdem der Administrator alle Einträge getestet hat, kann er unser Tool verwenden, um die Regeln, die die Funktionen brechen, anhand der fehlerhaften Einträge zu ermitteln. Letztendlich werden wir aber mehr Tests benötigen, um in Zukunft sicherere Systeme zu haben.

Literatur

[1] Center for Internet Security. CIS Benchmark for Windows 10, version 1909, 2022.

[2] Dietrich, C., Krombholz, K., Borgolte, K., and Fiebig, T. Investigating System Operators' Perspective on Security Misconfigurations. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2018), CCS '18, ACM, pp. 1272–1289.

[3] Dubslaff, C., Weis, K., Baier, C., and Apel, S. Causality in Configurable Software Systems. In *Proceedings of the 44th International Conference on Software Engineering* (New York, NY, USA, 2022), ICSE '22, Association for Computing Machinery, pp. 325–337.

[4] IBM CORPORATION. Cost of a Data Breach Report 2022, 7 2022.

[5] KUHN, D., WALLACE, D., AND GALLO, A. Software fault interactions and implications for software testing. *IEEE Transactions on Software Engineering 30*, 6 (2004), 418–421.

[6] KUHN, D. R., KACKER, R. N., AND LEI, Y. Practical Combinatorial Testing. Tech. Rep. NIST Special Publication (SP) 800-142, National Institute of Standards and Technology, Gaithersburg, MD, 10 2010.

[7] KUHN, R., KACKER, R., LEI, Y., AND HUNTER, J. Combinatorial Software Testing. *Computer 42*, 8 (2009), 94–96.

[8] LEI, Y., KACKER, R., KUHN, D. R., OKUN, V., AND LAWRENCE, J. 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.

[9] LEI, Y., KACKER, R., KUHN, D. R., OKUN, V., AND LAWRENCE, J. IPOG/IPOG-D: efficient test generation for multi-way combinatorial testing. *Software Testing, Verification and Reliability 18*, 3 (2008), 125–148.

[10] MANDL, R. Orthogonal Latin Squares: An Application of Experiment Design to Compiler Testing. *Commun. ACM 28*, 10 (oct 1985), 1054–1058.

[11] PEDREGOSA, F., VAROQUAUX, G., GRAMFORT, A., MICHEL, V., THIRION, B., GRISEL, O., BLONDEL, M., PRETTENHOFER, P., WEISS, R., DUBOURG, V., VANDERPLAS, J., PASSOS, A., COURNAPEAU, D., BRUCHER, M., PERROT, M., AND DUCHESNAY, E. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research 12* (2011), 2825–2830.

[12] RANDRIANAINA, G. A., TËRNAVA, X., KHELLADI, D. E., AND ACHER, M. 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.

[13] STÖCKLE, P., GROBAUER, B., AND PRETSCHNER, A. Automated Implementation of Windows-Related Security-Configuration Guides. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering* (New York, NY, USA, 2020), ASE '20, Association for Computing Machinery, p. 598–610.

[14] STÖCKLE, P., GROBAUER, B., AND PRETSCHNER, A. Sicherheitskonfigurationsrichtlinien effizient verwalten und umsetzen: Der Scapolite-Ansatz. In *Sicherheit in vernetzten Systemen: 29. DFN-Konferenz* (2022), A. Ude, Ed., DFN-CERT, BoD - Books on Demand.

[15] STÖCKLE, P., PRUTEANU, I., GROBAUER, B., AND PRETSCHNER, A. 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* (New York, NY, USA, 2022), CODASPY '22, Association for Computing Machinery, pp. 137–142.

[16] STÖCKLE, P., WASSERER, T., GROBAUER, B., AND PRETSCHNER, A. Automated Identification of Security-Relevant Configuration Settings Using NLP. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering* (Rochester, MI, USA, 2022), ASE '22, IEEE/ACM, Association for Computing Machinery.

[17] UL HAQUE, M., KHOLOOSI, M. M., AND BABAR, M. A. 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.

[18] VELEZ, M., JAMSHIDI, P., SIEGMUND, N., APEL, S., AND KÄSTNER, C. On Debugging the Performance of Configurable Software Systems: Developer Needs and Tailored Tool Support. In *Proceedings of the 44th International Conference on Software Engineering* (New York, NY, USA, 2022), ICSE '22, Association for Computing Machinery, pp. 1571–1583.

[19] WU, H., NIE, C., PETKE, J., JIA, Y., AND HARMAN, M. An Empirical Comparison of Combinatorial Testing, Random Testing and Adaptive Random Testing. *IEEE Transactions on Software Engineering 46*, 3 (2020), 302–320.

[20] YILMAZ, C., COHEN, M., AND PORTER, A. Covering arrays for efficient fault characterization in complex configuration spaces. *IEEE Transactions on Software Engineering 32*, 1 (2006), 20–34.

[21] YU, L., LEI, Y., KACKER, R. N., AND KUHN, D. R. ACTS: A Combinatorial Test Generation Tool. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation* (2013), pp. 370–375.